Een compiler en een parser in Prolog voor DPSG

van Horck, A.J.M.

Publication date:
1991

Document Version
Publisher's PDF, also known as Version of record

[redacted]

ITK

MEMO

[redacted]

ITK

STITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE

Een Compiler en een Parser in Prolog voor DPSG

ITK Memo No 9

Arthur J.M. van Horck

ITK
Katholieke Universiteit Brabant
Postbus 90153
5000 LE TILBURG
Telephone: +31 13 663060
Telefax: +31 13 663110
E-Mail: itk@kub.nl

Juli 1991

Uittreksel

Ik behandel hier een ontleedprogramma voor een fragment van het Nederlands, op basis van Discontinuous Phrase Structure Grammar (DPSG). DPSG is het grammaticaformalisme dat aan de Faculteit der Letteren wordt ontwikkeld in het kader van een onderzoek naar automatische verwerking van informatieve dialogen. Ten behoeve van het ontleedprogramma is ook een compiler voor de DPSG-grammaticaregels en voor het woordenboek geschreven. Ook deze behandel ik hier.

In de appendices treft U de definities van de invoerfiles en de source-code van alle programmatuur.

Inhoud

1. Inleiding

Een parser is een computerprogramma voor het analyseren van zinnen in een taal. Voor correcte zinnen levert een parser tevens een structuur op. Om dit te kunnen doen moet de parser beschikken over kennis, die is vastgelegd in drie afzonderlijke componenten, te weten een lexicon, een verzameling grammaticaregels en een algoritme. In het lexicon staan de relevante gegevens over de woorden (de terminale symbolen, lemmata, entries) van de taal. In de regels zijn de mogelijke zinsstructuren van de taal vastgelegd. Het algoritme bepaalt de volgorde waarin de regels en de lexicale elementen beschouwd worden.

In het geval van DPSG bevatten de grammaticaregels drie soorten informatie in drie hoofdbestanddelen van de regels.

Het eerste deel is van formele aard en legt vast welke terminale- en nonterminale symbolen een groter geheel kunnen vormen. Bijvoorbeeld,

```
(1) WW + NW --> Zin
```

betekent dat WW gevolgd door NW een Zin kan vormen.

Het tweede deel bevat zelf drie soorten informatie. De eerste soort legt vast onder welke omstandigheden de in het linkerdeel van de regel genoemde componenten mogen worden samengenomen. Hiervoor worden kenmerken gebruikt, zoals bijvoorbeeld FORM voor enkelvoud of meervoud. Een enkelvoudig werkwoord mag wèl met een enkel-, maar niet met een meervoudig naamwoord worden gecombineerd:

```
(2) WW.FORM = NW.FORM
```

De tweede soort kenmerkinformatie specificeert welke kenmerken van de samenstellende delen overgaan naar het grotere geheel, en de derde soort welke nieuwe kenmerken het geheel krijgt.

Het derde deel van de regel legt door middel van een formule in een logische taal vast hoe de betekenis van het geheel is opgebouwd uit de betekenissen van de samenstellende delen. Zie paragraaf 2.3.3., pp. 23 ff.

Het lexicon bestaat uit basiselementen, één voor elk woord van de taal. Elk entry bevat dezelfde drie soorten informatie. De informatie in deze drie delen is minder complex dan die in de regels omdat geen rekening hoeft te worden gehouden met informatie uit samenstellende delen. Het eerste deel bevat informatie over de syntactische categorie waartoe het woord behoort. Deel twee kent alleen condities van de derde soort en legt speciale kenmerken van het woord vast: is het woord enkel- of meervoudig, is het een voorkomen in de verleden of in de tegenwoordige tijd. Het derde deel bevat een definitie van de betekenis van het woord in een formule of constante in een logische taal. Zie paragraaf 2.3.3.

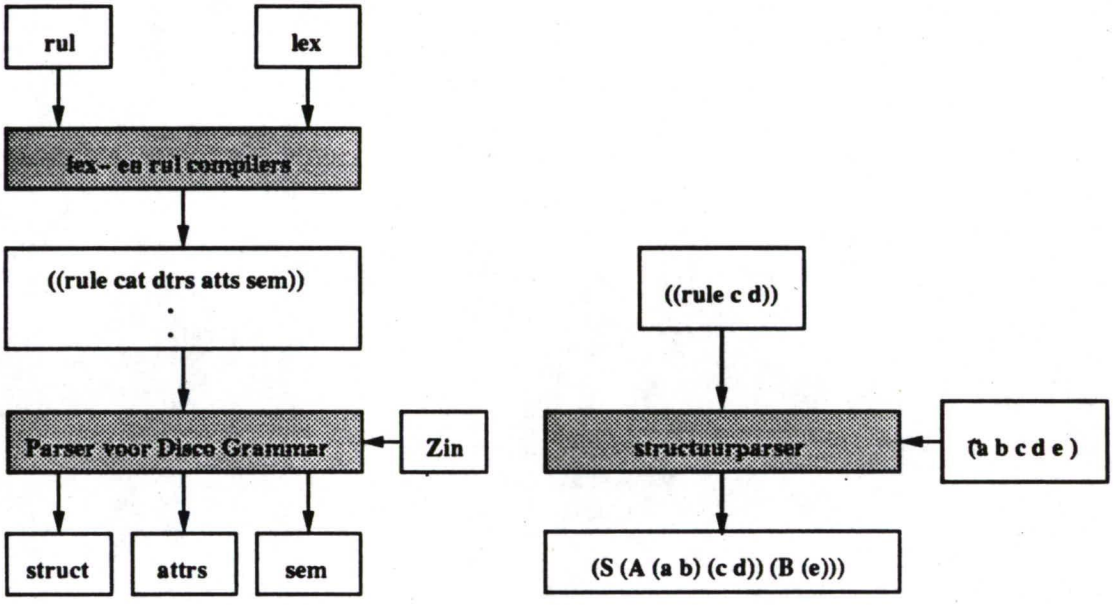
Een algoritme voor een parser is een voorschrift dat de keuze voor de regels en terminale symbolen stuurt bij de analyse van zinnen. Zo'n voorschrift kan worden getest door het op een computer te implementeren. Voor de implementatie van het algoritme dat in het kader van deze scriptie is ontworpen is gekozen voor de programmeertaal Prolog. Zie de hoofdstukken 3 en 5.

Het lexicon en de regels die in deze scriptie worden behandeld zijn opgeslagen in tekstbestanden. Deze bestanden bevatten de informatie die de parser nodig heeft. De basiseenheden van het lexicon en van de regelfile hebben een voorgeschreven vorm waarin de benodigde informatie systematisch wordt vastgelegd.

De formele definitie van het lexicon en van de regelfile is gegeven in de vorm van herschrijfregels, genoteerd in Backus Naur Formaat (BNF). Ook de BNF-definitie is als onderdeel van het scriptiewerk geformuleerd, waarbij uitgegaan werd van een bestaande gedeeltelijke definitie. Zie de paragrafen 2.1. en 2.2. en Bijlage A.

De vorm van het lexicon en van de grammaticafile is goeddeels bepaald door de behoefte van de menselijke gebruikers om de bestanden eenvoudig te kunnen wijzigen. De beschrijving van de regels en woorden moet soms worden aangepast aan nieuwe inzichten; regels en lexicon-entries worden toegevoegd of verwijderd. Deze bestanden zijn dan ook niet geschikt om rechtstreeks als invoer te dienen voor de parser. Allereerst moeten veranderde bestanden worden gecontroleerd op formele eisen die uit de definities voortvloeien. Verder is het tijdrovend en onhandig vanuit het oogpunt van de programmeur om bij herhaling informatie uit een bestand te halen dat zich in het achtergrondgeheugen bevindt, zoals op een diskette of op een band. De informatie uit de regels en de lexicon-entries moet, nadat is gecontroleerd of aan alle formele eisen is voldaan, worden opgeslagen in een van tevoren bepaalde datastructuur in het snel toegankelijke geheugen. Het programma dat zorgdraagt voor de controle en voor de transformatie naar een datastructuur heet een compiler. Zowel voor het regelbestand als voor het lexicon is als onderdeel van het scriptiewerk een compiler geschreven. Zie hoofdstuk 4.

In hoofdstuk twee behandel ik de BNF-notatie, alsmede de implementatietaal Prolog en relevante aspecten van het DPSG-formalisme. Hoofdstuk drie beschrijft de hoofdzaak van mijn scriptie-onderzoek, namelijk het ontwerp van een parser voor DPSG in Prolog. Daarvan komen de details in de hoofdstukken vier en vijf aan de orde. In hoofdstuk vier bespreek ik de datastructuur die voor de regels en voor het lexicon is ontworpen, en behandel ik de werking van de compilers. Hoofdstuk vijf beschrijft de parser. In hoofdstuk zes beschrijf ik hoe de compilers en de parser in de praktijk zijn getest. Tenslotte bevat hoofdstuk zeven enige conclusies.



Figuur 1: De gearceerde delen verwijzen naar de in deze scriptie besproken computerprogramma's

2. Formalismen voor de representatie van de grammatica

Voor de beschrijving van de invoerbestanden heb ik gebruik gemaakt van de Backus Naur notatie; de implementatie van alle programmatuur is gebeurd in (een dialect van) Prolog; de gebruikte grammatica is Discontinuous Phrase Structure Grammar. In dit hoofdstuk leid ik deze drie formalismen in.

2.1. BNF

BNF (Backus Naur Form) is een taal die is ontworpen om formele constructies en formele talen te beschrijven. BNF is door J.W. Backus gebruikt voor de definitie van de programmeertaal FORTRAN [Backus, 1967]; later is BNF onder andere gebruikt voor de definitie van de programmeertaal Pascal [Jensen & Wirth 1974, 1985]. Ik gebruik hier BNF om de vorm te definiëren van het lexicon en van de regelbestanden die op hun beurt dienen om de syntaxis van een fragment van het Nederlands te beschrijven.

In BNF worden herschrijfregels genoteerd van de vorm `A herschrijft als B gevolgd door C`. Daarbij zijn `B` en `C` òf zelf het resultaat van herschrijfregels òf eindsymbolen. In de herschrijfregels zijn twee soorten te onderscheiden: recursieve (3, 4 en 5) of niet recursieve regels (1, 2). Recursieve BNF-regels kunnen direct (3) of indirect recursief zijn: (4) is recursief via (5).

```
(1) valexpr              ::= '{' attvalset
(2) attvalset            ::= attval rest
(3) rest                 ::= ',' attval rest | '}'
(4) rest_van_const_sum ::= '+' constit_sum | '-->'
(5) constit_sum          ::= constit rest_van_const_sum
```

Ik hanteer in deze scriptie een enigzins uitgebreide en afwijkende notatie voor BNF. De de facto standaard biedt namelijk enige benodigde mogelijkheden niet. Zo bestaat er geen mogelijkheid om uitzonderingen aan te geven, zoals in de regel hieronder die uitdrukt dat een selector herschrijft als een willekeurige kleine letter, maar niet de letter `r`:

```
(6) selector ::= onderkast /'r'/
```

Ook bestaat er geen standaardnotatie voor 'nul of een keer', zoals in de volgende regel. Deze definieert `gco` als een `partlabel`, *optioneel* gevolgd door een `carry`, afgesloten met een `;`

```
(7) gco ::= partlabel [ carry ] ';'
```

De regel hieronder drukt uit dat `acln` bestaat uit één of meer voorkomens van `nac`:

```
(8) acln ::= { nac }
```

Meestal staan nonterminalen tussen de haakjes < en >. Ik doe dat hier niet; in plaats daarvan zet ik terminale symbolen tussen de enkele quotes '. Dit komt de leesbaarheid van de definities ten goede. Hieronder volgt een volledig overzicht van de gebruikte symbolen.

```
(9) x ::= y     x herschrijft als y
    { x }       x mag een of meer keer voorkomen
    [ x ]       x mag nul of een keer voorkomen (optie)
    x | y       x of y moet voorkomen (disjunctie)
    \* x *\     x is commentaar en hoort niet bij de definitie
    / x /       met uitzondering van x
    'x'         x is een terminaal symbool
```

Ik heb gekozen voor BNF als metataal voor de definitie van de invoerbestanden voor de compilers omdat het formalisme eenvoudig is en omdat het mogelijk wordt streng en eenduidig de vorm van correcte grammaticaregels en lexicon-entries vast te leggen. Zo'n definitie is nodig om controle op formele eigenschappen van de invoerbestanden mogelijk te maken bij automatische verwerking. Bovendien leent de vorm van herschrijfregels zich goed voor een snelle implementatie in Prolog.

2.2. PROLOG

Anders dan in traditionele, imperatieve, programmeertalen wordt in een Prolog programma niet vastgelegd hoe een probleem door de computer moet worden opgelost. In plaats daarvan wordt de logische structuur van het probleem vastgelegd; hoe de oplossing bereikt moet worden, wordt overgelaten aan het Prolog redeneermechanisme. Dat redeneermechanisme van Prolog is gebaseerd op het terugredeneren vanaf een conclusie naar de condities ervoor. Een probleem wordt zo gereduceerd tot steeds kleinere problemen, totdat er geen condities meer zijn. Het probleem is dan opgelost.

2.2.1. Het redeneermechanisme met resolutie en unificatie

Aan de hand van het volgende voorbeeld laat ik zien hoe in Prolog het terugredeneren vanaf een conclusie (backward reasoning) het effect heeft dat declaratieve stellingen leiden tot het uitvoeren van procedures.

```
(10) Alle mensen zijn sterfelijk.
     Socrates is een mens.
```

Deze twee zinnen kunnen in formele vorm worden uitgedrukt, door een conclusie en nul of meer condities. De eerste zin bijvoorbeeld heeft een conclusie: `X is sterfelijk`, en een conditie: `X is een mens`, waarbij `X` een variabele is die staat voor een willekeurig individu. De tweede zin heeft dezelfde struktuur: een conclusie, `Socrates is een mens`, en een triviale, lege conditie.

```
(11) X is sterfelijk
        als X is een mens

(12) Socrates is een mens
        als niets
```

(11) wordt in Prolog een regel genoemd, (12) een feit; beide zijn clauses: inferentieregels met één enkele conclusie en nul of meer condities. In deze clausale vorm kan verrassend veel kennis worden uitgedrukt.

Door middel van terugredeneren kan kennis die is uitgedrukt in clausale vorm omgezet worden in procedures. Zo leidt zin (11) hierboven tot een procedure die het probleem `Toon aan dat Socrates sterfelijk is` reduceert tot het aantonen dat Socrates een mens is. Dat laatste is zonder verdere condities vastgelegd in (12). In het algemeen werkt terugredeneren vanaf de conclusie van een clause (in (11): `X is sterfelijk`) naar de condities (in (11): `X is een mens`). In Prolog worden de condities strikt op volgorde van vermelding in de programmatekst afgewerkt.

(11) En (12) kunnen op deze manier worden opgevat als de twee 'procedures' (11') en (12"):

```
(11') Om aan te tonen dat X is sterfelijk
         toon aan dat X is een mens

(12') Om aan te tonen dat Socrates is een mens
         doe niets
```

De procedures (11') en (12') kunnen op twee manieren gebruikt worden: ten eerste kan specifieke informatie worden gevraagd met:

```
(13) ? X is sterfelijk
```

Dit staat voor de vraag: `Wie is er sterfelijk?`. Het antwoord is: `Socrates`. Dit antwoord wordt verkregen door het mechanisme van wegstrepen (resolutie) van de vraag tegen een mogelijk antwoord. Vraag en antwoord moeten daarvoor 'hetzelfde' zijn; zo is vraag (13) 'hetzelfde' als het eerste gedeelte van het antwoord (11). De vraag: `? X is sterfelijk` is hier weggestreept tegen het mogelijk antwoord: `X is sterfelijk (als X een mens is)`, gevonden in (11). Dit levert een nieuw vraag op, namelijk `Is er iemand die een mens is?`. Deze vraag wordt weggestreept tegen het mogelijk antwoord in (12): `Socrates is een mens`. Dit laatste mogelijke antwoord levert geen nieuwe vragen op; het wegstrepen van vragen tegen antwoorden leidt tot succes precies als er geen vragen meer overblijven en voor iedere vraag een mogelijk antwoord gevonden is.

Andersom kan een bewering worden geverifieerd met de meer concrete vraag:

```
(14) ? Socrates is sterfelijk
```

Hier is de vraag: `Is Socrates sterfelijk?`. Het antwoord volgens (11') is: `Als Socrates een mens is.` De vraag `Is Socrates een mens` vindt in (12') een antwoord: `Ja`. Het resolutiemechanisme levert hier geen nieuwe vragen op en het proces stopt succesvol.

Bij het beantwoorden van de algemene vraag (13) kreeg de variabele `x` de waarde `Socrates`: telkens wanneer een mogelijk antwoord voor een vraag wordt gevonden worden de overeenkomstige waarden uit het antwoord gesubstitueerd voor de variabelen in de vraag. Dit proces van substitutie ligt aan de basis van een meer algemeen proces, dat van unificatie. Bij unificatie wordt geprobeerd expressies aan elkaar gelijk te maken door uniform in beide expressies *tegelijk* te substitueren. Vraag en antwoord kunnen tegen elkaar worden weggestreept wanneer het mogelijk is ze door unificatie hetzelfde te maken.

2.2.2. De syntaxis van Micro Prolog

De software ten behoeve van deze scriptie is geschreven in LPA Micro Prolog versie 3.1. Prolog programma's en -queries worden opgebouwd uit atomaire formules met een predikaat-argumentstruktuur. Zo is in (15) `mens` het predikaat en `socrates` het argument:

```
(15) socrates is een mens
```

Een dergelijke predikaat-argumentstructuur heeft in Micro Prolog de vorm van een lijst. Het eerste element van die lijst is de predikaatconstante, en wordt gevolgd door de argumenten van het predikaat. De atomaire formule die correspondeert met (15) ziet er in Micro Prolog zo uit:

```
(15') (mens socrates)
```

Formules met een niet triviale conditie zoals (11) hebben de vorm

```
(16) conclusie als conditie1 en conditie2 en .. en conditien
```

In Micro Prolog zijn de logische connectieven `als` en `en` impliciet, en wordt de vorm (16) gerepresenteerd als een lijst bestaande uit de conclusie gevolgd door de condities zoals (16'):

```
(16') ((conclusie) (conditie1) (conditie2) .. (conditien))
```

```
(11') ((sterfelijk x) (mens x))
```

Alle variabelen beginnen in Micro Prolog met een underscore; (11') wordt dus genoteerd als (11''):

```
(11'') ((sterfelijk _x)
        (mens _x))
```

2.2.3. De algemene lijstnotatie

In Micro Prolog is het enige gestructureerde object de lijst. De eenvoudigste lijst is de lege lijst: `()`. Alle niet-lege lijsten zijn opgebouwd op basis van de lege lijst met behulp van de lijst constructor `|`. De lijst `(c)` bijvoorbeeld heeft de vorm `(c | ())`. Dit stelt de lijst voor die is opgebouwd door aan de lege lijst als eerste element het element `c` toe te voegen. Zo ook ontstaat `(b c)` door `b` vooraan toe te voegen aan `(c | ())`. De notatie `(t1 t2 .. (tn | t) .. )` is een verkorte schrijfwijze voor de expliciet geconstrueerde lijst (18)

```
(18) (t1 | (t2 | .. (tn | t ) .. ))
```

In (18) zijn zowel `t` als `t1 .. tn` zogenaamde termen: getallen, constantes, variabelen of zelf ook weer lijsten. Het symbool `|` kan gelezen worden als 'gevolgd door'.

Elke lijst waar variabelen in voorkomen is een *lijstpatroon*. Het patroon `( a b c | _rest)` is zo een patroon dat unificeert met elke lijst die als eerste drie elementen de constantes `a`, `b` en `c` heeft, eventueel nog gevolgd door een rest. Maar het patroon kan ook worden geunificeerd met elk ander lijstpatroon dat een lijst aanduidt waarin op de eerste drie plaatsen òf de overeenkomstige constanten òf ongebonden variabelen voorkomen. Het patroon `(_kop | _staart )` unificeert met elke lijst met minstens één element of met een lijstpatroon met als eerste element een vrije variabele.

2.2.4. Prolog en BNF

We hebben gezien in 1.1. dat BNF een schrijfwijze is voor het noteren van herschrijfregels van de vorm

```
(19) A ::= B C D
```

We wilen dit soort herschrijfregels operationaliseren in Prolog om automatisch te kunnen controleren of een regel- of lexiconbestand aan de formele eisen als beschreven in de BNF definitie voldoet. Dat wil zeggen dat we een programma willen schrijven dat iedere invoer precies zo herschrijft als door de BNF-regels is vastgelegd. We gaan dus *niet* een programma schrijven dat de BNF- notatie zelf herkent.

Herschrijfregel (19) kan worden gelezen als

```
(20) als de invoer B herkend is,
     vervolgens C,
     en vervolgens D,
     dan  mogen zij herschreven worden tot A

(21) A als
        B, gevolgd door
        C, gevolgd door
        D
```

B, C en D zijn hier als het ware condities voor A, net zoals de conditie `(mens _x)` voor de conclusie `(sterfelijk _x)` in (12″).

Omdat een Prolog definitie zelf altijd de vorm heeft `conclusie als condities` kan een BNF-herschrijfregel als een Prolog definitie worden geformuleerd. Met andere woorden: iedere regel van de vorm

```
(19) A ::= B C D
```

heeft als Prolog-definitie de vorm

```
(19') ((A)
       (B)
       (C)
       (D))
```

Het Prolog terugredeneermechanisme operationaliseert nu deze definitie door van de conclusie A terug te redeneren naar de condities B, gevolgd door C, gevolgd door D.

Met behulp van regels als (19) is de vorm vastgelegd van de regels uit het regelbestand voor de DPSG-grammatica. De kern van deze regels is overigens zelf van die vorm. De BNF-regels vormen de grammatica voor het formuleren van regels die op hun beurt de grammatica vormen voor een fragment van het Nederlands. Over deze laatste grammatica handelt de volgende paragraaf.

2.3. DPSG

DPSG is het grammaticaformalisme dat binnen het onderzoekprogramma van het werkverband Taal & Informatica wordt ontwikkeld voor de beschrijving van fragmenten van natuurlijke talen. De afkorting DPSG staat voor Discontinuous Phrase Structure Grammar.

Een 'phrase structure grammar' is een herschrijfgrammatica voor het beschrijven van structuren van zinsdelen (phrases). Het kenmerkende van DPSG is dat ook discontinue zinsdelen beschreven worden. Discontinue zinsdelen zijn zinsdelen die syntactisch- semantisch een eenheid vormen, maar die verspreid door de zin voorkomen zoals in de zinnen (22) t/m (24) hieronder.

```
(22) Ik heb een auto gekocht met vijf deuren.
(23) Ben Johnson is harder gegaan dan ooit tevoren.
(24) John talked, of course, about politics.
```

Het verschijnsel dat zinsdelen die syntactisch en semantisch bij elkaar horen verspreid in een zin voorkomen doet zich onder andere voor bij naamwoordelijke zinsdelen zoals 'een auto ... met vijf deuren' in (22), bij bijwoordelijke zinsdelen zoals 'harder ... dan ooit tevoren' in (23), en niet alleen in het Nederlands: (24).

In de inleiding hebben we gezien dat een DPSG-regel drie soorten informatie bevat, namelijk

- formele informatie in de PC-regel,
- linguïstische informatie in de PC-regel en in de gedeeltes van de DPSG-regel die betrekking hebben op attributen, en
- semantische informatie in de semantische regel.

In de PC-regel wordt de interne structuur van een phrase vastgelegd door aan te geven hoe één of meer constituenten samengenomen kunnen worden. De structuren die door een PSG aan phrases worden toegekend behoren allemaal tot de groep n-aire bomen, dat wil zeggen boomstructuren waar iedere knoop een willekeurig aantal vertakkingen mag hebben [van Eijck, 1986 p. 287 ff.]. Een PC-regel kent bovendien ook labels toe aan de knopen in die structuur. Zo'n label is de naam van de (syntactische) relatie tussen de constituenten. Bijvoorbeeld, de PC-regel

```
(25) NP + VP --> S
```

legt een relatie S tussen de twee constituenten NP en VP. Bovenstaande PC-regel eist dat er twee constituenten zijn in een bepaalde volgorde. Dat is formele, niet taalspecifieke informatie.

Daarnaast eist de regel dat de NP-relatie geldt tussen de constituenten van de eerste constituent, en de VP-relatie tussen de constituenten van de tweede constituent. Dit is linguïstische informatie.

De andere linguïstische informatie is vastgelegd in de gedeeltes van de regel die attribuutcondities, kenmerkoverdracht en nieuwe attribuutwaarden voor de topknoop beregelen.

De attribuut-informatie en de semantische informatie zijn specifiek voor de taal die door de grammatica beschreven wordt.

In DPSG worden voor de behandeling van discontinue phrases naast genoemde n-aire bomen ook *discontinue* n-aire bomen toegekend. Discontinue bomen zijn gedefinieerd in [Bunt 1987], en worden hieronder behandeld.

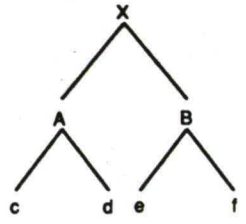
In de laatste paragrafen van dit hoofdstuk komen in het kort de centrale begrippen uit PSG (dominantie en precedentie in boomstructuren) aan bod en de aanpassingen ervan in DPSG aan structuren voor discontinue phrases. Voorts behandel ik de vorm van de bestanden waarin het lexicon en de regels zijn opgeslagen.

Hoofdstuk drie behandelt het skelet van de parser, namelijk dat gedeelte van de parser dat zich beperkt tot de formele beoordeling van strings. Hoofdstuk vier beschrijft de compilers voor het lexicon en voor de regels: voorwerk voor hoofdstuk vijf, waarin het skelet van de parser wordt uitgebreid met voorzieningen voor het verwerken van de linguïstische en semantische informatie uit de DPSG-regels.

2.3.1. Klassieke structuren in PSG's

In een PSG worden regels genoteerd voor het beschrijven van de structuur van zinnen van een taal. De structuur van die zinnen is tweedimensionaal. De twee dimensies noemen we *precedentie* en *dominantie*. Zo'n tweedimensionale structuur wordt meestal grafisch voorgesteld als een boomstructuur.

De precedentierelatie wordt bepaald door de volgorde van de terminale symbolen (de woorden) in de zin: deze worden geacht totaal geordend te zijn. Twee niet terminale knopen A en B behoren tot de precedentierelatie wanneer alle subknopen van A voorafgaan aan alle subknopen van B.



Figuur 2: Een voorbeeld van een klassieke boomstructuur

In de boom hierboven (figuur 2) gaat A vooraf aan B (genoteerd als A < B) omdat alle subknopen van A (c en d) voorafgaan aan alle subknopen van B (e en f).

Een van precedentie afgeleid begrip is *adjacentie*, namelijk precedentie zonder tussenliggende knopen. Zo zijn in figuur 2 c en d, d en e, e en f en A en B adjacent of, met een ander woord, *buren*. Het adjacentie-begrip is noodzakelijk voor het formuleren van regels. De regel

```
(27) c + d --> A
```

betekent dat de knoop c gevolgd door de knoop d (in die volgorde en zonder tussenliggende knopen) herschreven mag worden tot A. De + operator duidt adjacentie (een buurpaar) aan.

Behalve precedentie is in regel (27) ook de dominantierelatie binnen deze deelboom uitgedrukt: A domineert c en d. Dit wordt wel genoteerd als A D c en A D d. Daarin betekent D *Domineert direct*. Elke structuur heeft een enkele topknoop, die direct (D) of indirect (genoteerd als D*) elke andere knoop in de structuur domineert. Zo geldt in figuur 2 onder andere X D* f, ofwel X domineert f indirect (namelijk via de reeks X D B, B D f). Verder is het zo dat elke knoop, behalve de topknoop (die immers door geen andere knoop gedomineerd wordt), door precies één knoop direct wordt gedomineerd.

Dominantie en precedentie zijn wederzijds exclusief: twee knopen behoren *of* tot de dominantierelatie, *of* tot de precedentierelatie, maar niet beide. Zo geldt in voorbeeld

figuur 2 wel c < d, maar niet c D d of d D c, en omgekeerd geldt dat A D c, maar niet A < c of c < A.

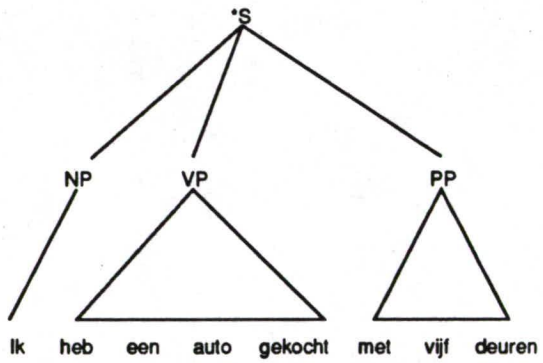
2.3.2. Discontinue structuren in DPSG's

In de inleiding is al aangeduid dat in natuurlijke talen verschillende vormen van discontinuïteiten voorkomen. In PSG's staan ons geen middelen ter beschikking om deze discontinuïteiten uit te drukken in de boomstructuur die aan een zin wordt toegekend omdat de definitie van de buurrelatie daarvoor te restrictief is. De relatie 'verre buur zijn' kan niet worden uitgedrukt. De reden om voor phrases met discontinuïteiten ook discontinue boomstructuren toe te willen staan is er een van semantische aard. Ik illustreer dat aan de hand van de volgende twee voorbeelden.

In de zin (22) ligt het voor de hand de phrase een auto met 5 deuren als een semantische eenheid te beschouwen. Met 'klassieke' middelen is het niet mogelijk een structuur te bereiken waarin deze ook zo voorkomt.

(22) Ik heb een auto gekocht met 5 deuren

Twee mogelijkheden dringen zich in de traditionele benadering op: Ten eerste: beschouw met 5 deuren als modifier bij de gehele zin, als in figuur 3. Ten tweede:



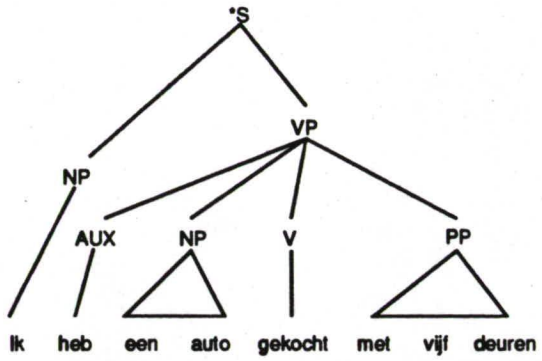
Figuur 3: PP als zins-modifier

beschouw met 5 deuren als modifier bij het werkwoord (figuur 4).

Geen van beide mogelijkheden komt ook maar in de in de buurt van hetgeen we willen, namelijk een modifier bij de NP een auto. In figuur 5 is een boomstructuur aangegeven zoals die mogelijk is wanneer we discontinue structuren toestaan.

Het tweede voorbeeld betreft een veel eenvoudiger geval, dat van de ambigue zin Jan heeft Marie gekust (figuur 6).

Figuur 6 is een weergave met behulp van een PSG van een mogelijke structuur voor de zin. Wanneer we nu op grond van deze structuur betekenis willen gaan toekennen aan deze zin kunnen we slechts de interpretatie bereiken waarbij Jan de

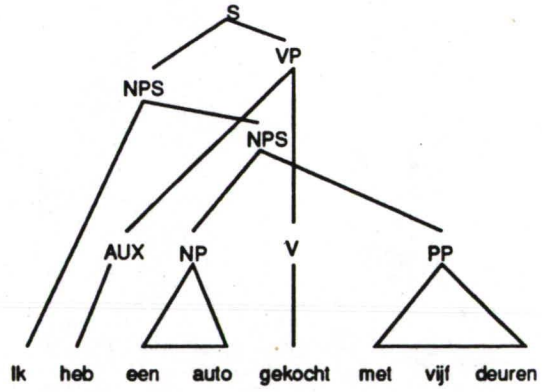


Figuur 4: PP als werkwoords-modifier

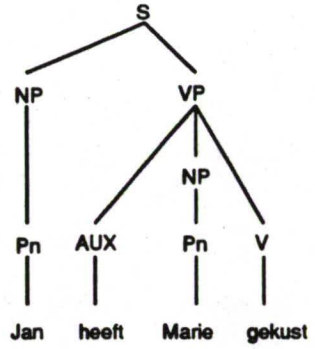
subjectrol heeft en `Marie` de objectrol. Willen we ook de (gemarkeerde) omgekeerde interpretatie krijgen dan moeten ook in de boomstructuur `Jan` en `Marie` omgewisseld worden. Traditioneel wordt dit bereikt door een transformatieregel toe te passen die beide eindsymbolen in de boomstructuur verwisselt (figuur 7 op pagina 19. De boom*structuur* is weliswaar dezelfde, de *boom* is een andere: de terminale symbolen zijn in figuur 6 hieronder en in figuur 7 op pagina 19 verschillend gerangschikt.

Semantische interpretatie zien we hier als het toekennen aan een zin van een *semantische representatie*, een weergave in een logische taal, waarvan de semantiek reeds gedefinieerd is. Nu kunnen `Jan` en `Marie` , logisch gezien, beschouwd worden als argumenten bij het predikaat `kussen`: KUSSEN(Jan Marie). Het lijkt aantrekkelijk deze argumenten in één constituent samen te nemen, als in figuur 8 hieronder:

De twee NP's zijn hier samen ondergebracht in een complexe NP- structuur (NPS, Noun Phrase Sequence). We kunnen nu `Jan` en `Marie` binnen de semantische representatie van die NPS naar hartelust permuteren, zonder de boomstructuur die bij de



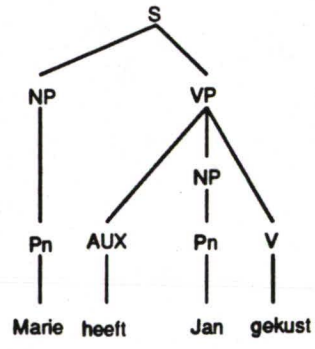
Figuur 5: Een boom met discontinue structuren



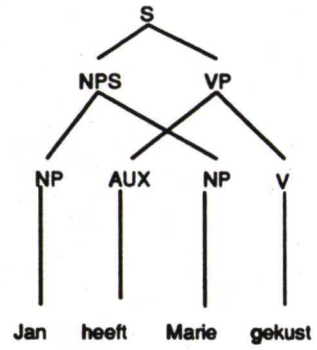
Figuur 6: Een ambigue zin

zin hoort te hoeven veranderen, en zonder de terminale symbolen in de zin anders te hoeven rangschikken. Met andere woorden: we kunnen met één enkele syntactische structuur meerdere interpretaties opleveren. Een nadeel is dat we nu een boom hebben met gekruiste takken. Zo'n boom kan door de manier waarop in PSG's dominantie en precedentie zijn gedefinieerd helemaal niet bestaan: in klassieke bomen moeten immers alle subknopen van een knoop tussen de meest linkse en de meest rechtse subknoop van die knoop door die knoop worden gedomineerd. In figuur 8 komt `heeft` voor tussen `Jan` en `Marie`, subknopen van de `NP`S, maar `heeft` wordt niet gedomineerd door die `NP`S (maar door de `VP`!). Zo ook komt `Marie` voor tussen `heeft` en `gekust`, subknopen van de `VP`, maar `Marie` wordt niet door de `VP` gedomineerd maar door de `NP`S.

Om boomstructuren met discontinuïteiten mogelijk te maken moeten de precedentierelatie en het adjacentiebegrip (in de regels uitgedrukt met +) opnieuw, lees: ruimer, geformuleerd worden.



Figuur 7: De boom uit figuur 6 na transformatie



Figuur 8: De boom uit figuur 6 in een analyse met discontinue constituenten

De precedentierelatie word in DPSG geformuleerd als volgt:

(28) Een nonterminale knoop x in de verzameling knopen van een boom staat links van een knoop y in de boom desda de meest linkse dochter van x staat links van de meest linkse dochter van y

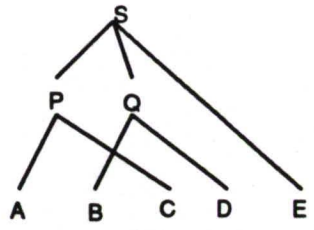
Omdat de terminale knopen in een boom totaal geordend zijn wordt er in de definitie niet over gesproken. Het begrip 'meest linkse dochter' (zie [Bunt '88]) wordt hier niet nader uitgewerkt.

Het adjacentiebegrip wordt vervangen door het ruimere begrip van een *n-plaatsige buurrelatie.* Deze is gedefinieerd als volgt:

(29) Een reeks (a, b, ..., n) is een n-plaatsige buurreeks desda
(1) elk paar (i, j) in de reeks is
- of een buurpaar
- of verbonden door een reeks buurparen waarvan elk element direct of indirect gedomineerd wordt door een element van de reeks (a, b, ..., i);
(2) de elementen in de reeks geen constituenten gemeenschappelijk hebben

In figuur 9 hieronder is P Q E bijvoorbeeld een buurreeks omdat P Q een buurpaar is en Q en E verbonden zijn via de reeks buurparen Q-C-D-E. Daarbij zijn C en D constituenten van respectievelijk P en Q. P B C is in figuur 9 geen buurreeks, omdat P en C immers C gemeenschappelijk hebben: C is een subconstituent van P.

De regels die gebruikt zijn voor het construeren van de boom in figuur 9 volgen hieronder:



Figuur 9: Een illustratie van de n-plaatsige buurrelatie

```
(30) A + [B] + C --> P
     B + [C] + D --> Q
     P + Q + E --> S
```

De eerste regel drukt dus uit dat A gevolgd door B gevolgd door C herschreven mag worden tot P, maar dat B niet door P wordt gedomineerd. [B] geeft aan dat B als interne context van P moet worden gezien. In de regels duidt de + zoals gezegd adjacentie aan. De laatste regel drukt uit dat P gevolgd door Q gevolgd door E herschreven mag worden tot S.

2.3.3. Enige details van het DPSG-formalisme

In deze paragraaf behandel ik nog de vorm van de PC-regels, de vorm van het lexicon en van de regelfile, en behandel ik één van de criteria die van belang zijn bij het formuleren van PC-regels.

Over de vorm van de PC-regels. In DPSG worden de regels die de boomstructuren genereren (de PC-regels) in een aangepaste vorm genoteerd, met selectoren a, b, c, enzovoorts, en r voor het resultaat:

```
(31) a:A + [b:B] + c:C --> r:P
     a:B + [b:C] + c:D --> r:Q
     a:P + b:Q + c:E --> r:S
```

De zogenaamde selectoren zijn toegevoegd om in de uitgebreide regels, waarin ook attribuutcondities, carry-over en semantiek behandeld worden, eenduidig te kunnen verwijzen naar constituenten. In een regel als

```
(32) a:NPS + [b:AUX] + c:NPS --> r:NPS
```

zou het zonder selectoren niet eenvoudig zijn te eisen dat het attribuut FORM van de eerste NPS de waarde SING moet hebben, dat attribuut van de tweede de waarde PLUR moet hebben, en dat de resulterende topknoop voor dat attribuut de waarde PLUR moet krijgen, bijvoorbeeld.

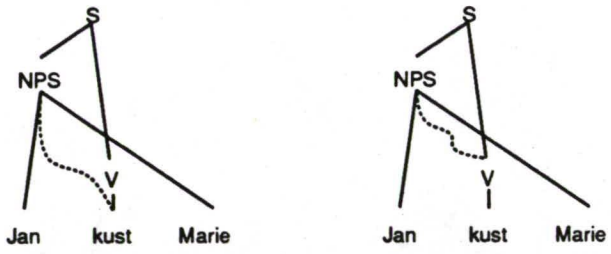
Verder kunnen in de grammatica regels voorkomen met optionele constituenten. Deze staan tussen (en). Met behulp van dergelijke constituenten kunnen meerdere regels tot een enkele regel worden samengevoegd. Zo is de regel (33) hieronder

```
(33) a:A + (b:B) + (c:C) + d:D --> r:X
```

een combinatie van de vier regels (39)

```
(34) a:A + b:D --> r:X
     a:A + b:B + c:D --> r:X
     a:A + b:C + c:D --> r:X
     a:A + b:B + c:C + d:D --> r:X
```

Contextpijlen zijn hulplijnen. Beschouw de volgende twee bomen die dezelfde zin beschrijven:



Figuur 10: Twee analyses voor 'Jan kust Marie'

De bomen in figuur 10 verschillen daarin dat in de eerste boom de NPS de terminaal kust als contextconstituent heeft, maar de tweede boom als contextconstituent de topknoop V. De bijbehorende (vereenvoudigde) grammatica's noemen dan ook in het ene geval de terminaal kust en in het andere geval de nonterminaal V als contextelement:

```
(35) S   --> NPS V               S   --> NPS V
     NPS --> Jan [kust] Marie    NPS --> Jan [V] Marie
     V   --> kust                V   --> kust
```

Als beide regelsets tot een grammatica worden gecombineerd:

```
(36) S   --> NPS V
     NPS --> Jan [kust] Marie
     NPS --> Jan [V] Marie
     V   --> kust
```

mogen we dan zeggen dat de zin Jan kust Marie structureel ambigu is omdat er op het eerste gezicht twee structuren toegekend kunnen worden? Neen, dat is niet zo; om dat te kunnen zien brengen we de definitie van een boomstructuur in herinnering.

Een boom is een verzameling knopen waarop twee relaties zijn gedefinieerd: dominatie en lineaire ordening. De dominantie- en ordeningsrelaties in de twee voorbeeldbomen in figuur 10 zijn identiek, zodat de bomen formeel niet van elkaar verschillen.

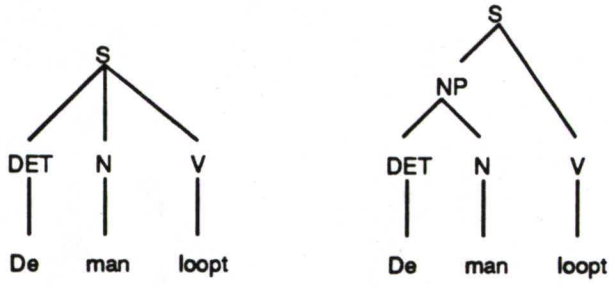
Voor de dominantierelatie is dat als volgt in te zien: omdat tussen knopen en hun interne contextconstituenten geen enkele dominantierelatie is gedefinieerd zal de keuze tussen KUST en V als interne contextconstituent van de NPS de dominantierelatie niet veranderen.

Voor de links-rechtsrelatie ligt dat anders. Elk contextelement onder een knoop staat links of rechts van die knoop. De keuze van de contextelementen is daarom van invloed op de links- rechtsrelatie in de boomstructuur.

Toch wordt hier de links-rechtsrelatie niet beïnvloed. In het begin van deze paragraaf hebben we gezien dat een topknoop (in dit geval V) en zijn meest linkse dochter (in dit geval `kust`) beide in dezelfde links-rechtsrelatie staan tot de andere knopen in de boom. Dat is namelijk altijd het geval tussen een knoop en zijn meest links dochter. De keuze tussen een knoop en zijn meest linkse dochter als contextelement verandert de links-rechtsrelatie in een boom dus niet. Dit volgt met inductie uit definitie 28 (pagina 22).

Dit illustreert dat de contextpijlen in een getekende boom slechts hulplijnen zijn en niet behoren tot de eigenlijke boomstruktuur.

Er is wel een goed argument om de voorkeur te geven aan de tweede boom boven de eerste. Dat laat ik zien aan de hand van de volgende twee bomen. Hoewel daarin geen contextconstituenten voorkomen, en de bomen een verschillend aantal knopen hebben, illustreren ze wel het argument op grond waarvan in figuur 10 de tweede boom de voorkeur verdient boven de eerste.



Figuur 11: Twee analyses van 'De man loopt'

Waarom is de eerste boom hierboven niet acceptabel? Deze boom is niet acceptabel omdat daarin niet de (intuïtief) juiste functionele eenheden zijn onderscheiden. Het element DE heeft geen functionele rol in de globale zin, maar wordt toch als directe subconstituent ervan genoemd. DE MAN daarentegen is wel een functionele eenheid in die zin, maar wordt niet als zodanig aangemerkt in de eerste boom.

Het onderscheiden van semantisch functionele eenheden in een zin moet bovendien zo algemeen mogelijk gebeuren. Dat geldt ook voor eenheden die als contextelement voorkomen. In figuur 10 wordt in de tweede boom de meer algemene eenheid v als contextconstituent gekozen. Daarom verdient deze boom de voorkeur boven de tweede in figuur 10.

Wanneer contextelementen geen rol spelen zal een keuze vanzelf tot stand komen als de zin niet ambigu is, omdat de verschillende varianten verschillende bomen zijn. Dat zal altijd zo zijn zolang contextelementen geen rol spelen, want ieder verschil in constituentkeuze heeft dan invloed op de dominantierelatie.

De keuze voor de meest algemene functionele eenheid is formeel niet dwingend voor contextconstituenten, omdat de minder algemene varianten formeel dezelfde bomen zijn, met dezelfde dominantie- en ordeningsrelaties. De ontwerper van een grammatica zal zich daarom nog beter bewust moeten zijn van het methodologische principe van de keuze voor de meest algemene functionele eenheid, omdat het formalisme hier ruimte laat.

Het lexicon. Tot hier is steeds sprake geweest van 'het lexicon'. In het geval van DPSG bestaat het lexicon uit drie afzonderlijke bestanden. Het eerste is het NEDELF-lexicon, en bevat informatie over de woorden van de taal alsmede een vertaling van de woorden van de taal in ELf. Het tweede lexicon is het ELFELR-lexicon. Dit bevat een vertaling van alle ELf-constanten in ELr, waarbij alle ambiguiteiten worden weggezuiverd. Het ELRCON-lexicon tenslotte bevat de types van alle constanten uit ELr. Alleen het eerste lexicon is voor deze scriptie van belang, de beschrijving van het lexicon strekt zich daarom ook niet uit tot de andere twee. Waar sprake is van 'het lexicon' bedoel ik steeds het NEDELF-lexicon.

De lexiconfile bestaat uit een (optioneel) commentaar, tussen { en }. In dat commentaar staan de laatste wijzigingen aan de file vermeld, door wie deze zijn aangebracht, en zonodig wanneer en waarom. Daarna komen de entries. Dat moet er minstens één zijn. De reeks wordt afgesloten met het symbool `end-of-lex`.

Hieronder ziet u een voorbeeld van een entry in het lexicon. Aan de hand hiervan zal ik het formaat van die entries beschrijven.

```
(37) HET      \2  CENTRALDET
                  FORM SING GENDER NEUTR *
                  constant CRo ;
                  CENTRALDET
                  FORM MASS GENDER NEUTR *
                  constant CRo ;
```

Een entry in het lexicon bestaat uit vijf componenten. Allereerst zien we het woord van de taal dat beschreven wordt: (HET hierboven). Dat wordt gevolgd door een getal (2 hierboven) dat aangeeft hoeveel interpretaties dat lemma heeft. Dan volgt de syntactische categorie van het woord (CENTRALDET in het voorbeeld), en vervolgens een reeks van nul of meer attribuut-waarde paren (FORM SING GENDER NEUTR in de eerste interpretatie in het voorbeeld hierboven). Het einde van die reeks wordt aangegeven met een *. Als laatste onderdeel treffen we een expressie in ELf aan die de betekenis van het woord op formeel niveau vastlegt. In het voorbeeld is die betekenis vastgelegd met behulp van de constante CRo, die verwijst naar de context van de zin. Het einde van elke interpretatie wordt aangegeven met een ;. Als een woord slechts één interpretatie heeft ontbreekt het getal, en als een woord meerdere interpretaties heeft wordt dat woord zelf alleen bij de eerste interpretatie genoemd.

De BNF definitie van de lexiconfile is rechttoe, rechtaan, en levert vanwege de eenvoud geen problemen op. De complete definitie van het lexicon treft u aan in bijlage 1.

De regelfile. De regelfile bestaat, net als de lexiconfile, uit een commentaar tussen { en }, gevolgd door minstens één regel. Elke regel heeft een naam, en wordt afgesloten met een 'end-of-rule' symbool: -*-. Op de naam volgt een kort commentaar, meestal alleen de naam van de resulterende topknoop met een index. Elk deel van de regel wordt voorafgegaan door een beschrijvend label, gevolgd door een : en wordt afgesloten met een ;.

```
(38) E1 { NPCENTRE1 }

     PC Rule              : a:CENTRALDET + b:NOM --> r:NPCENTRE ;

     Local attribute
     conditions           : a.form = [plur] ;

     Global attribute
     conditions           : a.form = b.form ;

     Carry over of
     attribute values     :                  b   --> r

     Incidental
     carry over           : ;

     New attribute
     values               :                  r.defness := def ;

     Semantic rule        : a' + b' --> selection
```

```
                                    parameter b
                                    abstraction
                                        variable x anytype
                                        application
                                           parameter b
                                           variable x ;

    -*-
```

Het eerste deel, de PC-regel, is hierboven in 2.3. al aan de orde gekomen. In het tweede deel worden de locale attribuutcondities genoteerd in de vorm

```
(46) selector.attribuut RELATIE waarde
```

`selector` Is daarbij een van de selectoren uit het linkerdeel van de PC-regel. `attribuut` Is een voor de bij de `selector` horende categorie relevant attribuut. `RELATIE` is een relatie uit het rijtje `IN`, `CONTAINS`, `NOT IN`, `NOT CONTAINS`, = en <>. `waarde` Is een voor het in de regel genoemde attribuut relevante waarde. De vorm van die waarde is afhankelijk van het attribuut, en kan een lijst zijn, een set of een atomaire waarde. De punt tussen de selector en het attribuut vindt haar oorsprong in de oorspronkelijke Pascal implementatie en duidt het veld `attribuut` van de recordvariabele `selector` aan.

In de notatie van de globale attribuutcondities wordt geen specifieke waarde geeist, maar wordt gelijkheid afgedwongen van de waarde van hetzelfde attribuut bij verschillende subknopen:

```
(47) selector1.attribuut = selector2.attribuut
```

Attribuutwaarde overdracht wordt op twee manieren gespecificeerd. Bij de globale overdracht gaan alle kenmerken met hun waarde over op de topknoop:

```
(48) selector --> r
```

Bij de incidentele overdracht gaan slechts een of een paar kenmerken van een constituent over naar de topknoop:

```
(49) selector.attribuut --> r.attribuut
```

In de toekenning van nieuwe attribuutwaarden aan de resulterende topknoop wordt de Pascal assignment operator gebruikt:

```
(50) r.attribuut := waarde
```

Het laatste deel van een DPSG regel bevat een uitgeschreven ELf expressie.

Al de delen, behalve de PC-regel en de semantische regel, mogen leeg zijn: in dat geval volgt de afsluitende ; direct op de : die het label afsluit.

De volledige beschrijving in BNF-notatie treft U aan in bijlage A.

3. Het herkennen van algemene discontinue structuren in Prolog

In dit hoofdstuk wordt een herkenner voor discontinue bomen ontwikkeld. Deze vormt de basis voor de parser uit hoofdstuk vijf die de linguïstische informatie uit de DPSG-regels combineert met de formele informatie uit de PC-regels.

De bewijsmethode van top-down backtracking resolutie die door Prolog wordt gehanteerd maakt het betrekkelijk eenvoudig een parserschema op te zetten voor PSG's dat die strategie gebruikt. Hoewel er voor PSG's veel betere en veel efficiëntere algoritmes bestaan moet zo'n top down, depth first, van links naar rechts door de invoerstring werkend algoritme in het kader van deze scriptie als voldoende worden beschouwd.

In de eerste paragraaf wordt uiteengezet hoe zo'n top-down parserschema voor 'klassieke bomen' in Prolog kan worden gedefinieerd. Vervolgens laat ik in de tweede paragraaf zien welke uitbreidingen nodig zijn om discontinue bomen te kunnen behandelen.

3.1. Het parseren van klassieke structuren in Prolog

We kunnen herschrijfregels van de vorm

```
(1) topknoop --> dochter1 dochter2
    a:dochter1 + b: dochter2 --> r:topknoop
```

in Micro-Prolog noteren als het predikaat (2):

```
(2) ((regel _knoop _lijst-van-dochters))
```

Instanties van (2) zijn bijvoorbeeld

```
(3) ((regel S (NP VP) ))
    ((regel NP (Jan) ))
    ((regel NP (DET (opt ADJS) NOUN) ))
```

Wanneer we regels zo representeren kunnen we een eenvoudige herkenner in Prolog schrijven. We geven het predikaat de naam `parse`. Dit programma moet woord voor woord een invoerstring analyseren, en levert succes op wanneer alle woorden van de invoerstring gebruikt zijn bij het herkennen van een bepaalde knoop. Hieruit volgt dat `parse` drieplaatsig moet zijn:

```
(4) ((parse _knoop _invoerstring _resultaat))
```

Alle argumenten zijn lijsten. Bij de aanroep van het programma geven wij `_knoop` de gewenste topknoop als enig element. `_invoerstring` wordt de lijst met woorden die 'in die knoop gepast moeten worden'; `_resultaat` moet leeg zijn aan het eind van de parse: we willen aan het eind immers geen onverwerkte reststring overhouden. Omdat in Prolog geen assignment bestaat kan `_invoer` niet leeg gemaakt worden; daarom hebben wij `_uitvoer` nodig om dat te doen. De vraag aan het programma kan er bijvoorbeeld (8) zo uitzien:

```
(5) ?((parse (S) (Jan loopt) () ))
```

Dit betekent: is de hele lijst `(Jan loopt)` te herkennen als een frase met topknoop S? Hier is het programma:

```
(6) ((parse () _invoer _invoer))
    ((parse (_knoop|_knopen) (_term|_rest) _resultaat)
       (regel _knoop (_term))
       (parse _knopen _rest _resultaat))
    ((parse (_knoop|_knopen) _woorden _resultaat)
       (regel _knoop _dochters)
       (parse _dochters _woorden _tussenres)
       (parse _knopen _tussenres _resultaat))
```

Men een grammatica en een lexicon van de vorm die hierboven is beschreven:

```
(7) ((regel S (NP VP) ))
    ((regel NP (Jan) ))
    ((regel VP (loopt) ))
```

zal de parser `(Jan loopt)` als volgt herkennen als een zin van de beschreven taal:

- Allereerst zal het systeem proberen de query

  ```
  (8) ?((parse (S) (Jan loopt) ()))
  ```

  te unificeren met de eerste clause van het programma. Dit mislukt: de eerste parameter `(S)` komt niet overeen met de lege lijst `()` als eerste parameter in de eerste clause.

- Vervolgens wordt de tweede clause geprobeerd. Dit levert in eerste instantie succes op, omdat `(S)` een instantie is van `(_knoop|_knopen)`, `(Jan loopt)` een instantie van `(_woord|_woorden)`, en `()` een instantie is van de (nog ongebonden) variabele `_resultaat`. Bij het oplossen van de eerste conditie loopt het echter spaak: er is in de database geen regel te vinden die `(S)` herschtijft als `(Jan)`. De executie van de tweede clause houdt hier op en ...

- de derde clause wordt geprobeerd. `(S)` matcht hier met `(_knoop|_knopen)`, `(Jan loopt)` matcht met `_woorden`, en `()` met de variabele `_resultaat`. Wanneer de eerste conditie wordt geprobeerd wordt er nu wèl een regel gevonden die `(S)` herschrijft, namelijk als `(NP VP)`. Zo wordt de nieuwe vraag

  ```
  (9) (parse (NP VP) (Jan loopt) _tussenresultaat)
  ```

  uit de eerste recursieve aanroep van `parse` in de derde clause van `parse`.

- De eerste clause wordt weer geprobeerd, weer zonder resultaat, maar de tweede clause levert nu wel succes op: de eerste conditie van de tweede clause levert nu een regel die NP herschrijft als `Jan`. NP wordt uit de eerste lijst gezet, `Jan` uit de tweede, en
- De tweede conditie wordt ingegaan met de variabelen geïnstantieerd als

```
(10) (parse (VP) (loopt) _tussenresultaat)
```

Ook hier levert de eerste conditie van de tweede clause een regel op: de regel die een VP herschrijft als `loopt`. VP en `loopt` worden weer verwijderd uit de lijsten waar ze in stonden en de tweede recursieve conditie van clause twee wordt nu ingegaan met

```
(11) (parse () () _resultaat)
```

waarin de eerste lege lijst aangeeft dat er geen constituenten meer gevraagd worden.

- Nu wordt wel gematched met de eerste clause: de eerste `()` uit de call matcht met de eerste lege lijst van de clause, de tweede ook, en de (nog steeds ongeïnstantieerde) variabele `_tussenresultaat` krijgt hier door unificatie de waarde `()`. De tweede call uit de derde clause is nu afgewerkt. Rest nog de derde call van die derde clause: De instanties zijn

```
(12) (parse () () ())
```

en dat matcht weer met de eerste clause. Het laatste tussenresultaat was `()`, en dat was bij de vraag ook geeist door aan resultaat de waarde `()` mee te geven. het enige verschil met de vorige match is dat `_resultaat` nu al een waarde heeft: `()`.

Het proces is ten einde omdat er

1. geen onafgewerkte condities overzijn en
2. omdat de invoerstring op is.

De herkenner voor continue structuren is nu klaar; een parser levert echter, behalve een antwoord op de vraag of een zin tot de in de grammatica beschreven taal behoort ook een structuur op voor correcte zinnen. Met een kleine aanpassing aan de parse-clauses is dat te bereiken:

```
(13) ((parse () _invoer _invoer ()))
     ((parse (_knp|_knpn) (_trm|_rst) _rsltt ((_knp |(_trm))|_strcs))
        (regel _knp (_trm))
```

```
    (parse _knpn _rst _rsltt _strcs))
((parse (_knp|_knpn) _wrdn _rsltt ((_knp |_strcd)|_strcs)
    (regel _knp _dchtrs)
    (parse _dchtrs _wrdn _tssnrs _strcd)
    (parse _knpn _tssnrs _rsltt _strcs))
```

Zoals U ziet is een vierde parameter aan elke `parse`-clausule toegevoegd. Daarin wordt middels unificatie steeds de structuur van de constituenten met de huidige knoop `_knp` vooraan in de lijst met de structuur geplaatst. Wanneer die knoop een terminale knoop is wordt de knoop met zijn terminale symbool `(_knp | (_trm))` vooropgeplaatst, anders de knoop met de structuur van de dochters: `(_knp |_strcd)`.

3.2. Het parseren van discontinue structuren in Prolog

Hierboven hebben we gezien hoe we regels voor 'klassieke' n-aire boomstructuren kunnen representeren en herkennen. Behalve deze 'klassieke' regels kent DPSG (zie 2.3.1) echter ook regels waarin interne contextconstituenten worden gespecificeerd als in

```
(14) a:NPS + [b:AUX] + c:NPS --> r:NPS
```

Daarin specificeert `[b:AUX]` een constituent met het label `AUX` als tweede 'constituent' in een reeks die tot een `NPS` mag worden herschreven. We kunnen interne contextelementen in de Prolog-regels representeren als lijsten van de vorm (15):

```
(15) ( [ _cat ] )
```

Rekening houdend met de representatie voor herschrijfregels uit de vorige paragraaf kunnen we regel (14) in Prolog uitdrukken als

```
(16) ((regel NPS (NPS ([AUX]) NPS) ))
```

In 2.3.1. hebben we gezien dat contextelementen niet gedomineerd worden door de phrase waarin ze voorkomen. Zo wordt de `AUX` in (14) niet gedomineerd door `r:NPS`. Dit betekent dat bij het herkennen van `r:NPS` de `AUX` bewaard moet blijven omdat die elders in de boom gedomineerd wordt en dus later in het herkenningsproces weer nodig is. Hoewel `a:NPS` en `c:NPS` geen interne contextelementen zijn in (14) moeten ze eveneens bewaard worden. Zowel `a:NPS` als `c:NPS` kunnen immers interne context zijn in bepaalde phrases elders in de structuur.

Met andere woorden, de introductie van contextelementen brengt met zich mee dat een parser als nieuw element een 'bewaarstructuur' nodig heeft.

Aan de datastructuur voor het bewaren van herkende phrases moeten twee eisen worden gesteld. De volgorde waarin de phrases in de regels worden genoemd moet herkenbaar blijven omdat die volgorde van belang is voor de regels die deze phrases

elders weer als constituent noemen. Daarnaast moet uit de datastructuur blijken of een constituent als contextelement of als echte constituent herkend was omdat deze laatste niet weer als echte constituent genoemd mag worden.

Dit betekent dat de volgorde van verwerking ook weerspiegeld moet worden in de datastructuur, èn dat de elementen erin gemarkeerd moeten worden naar hun status. Contextelementen worden gemarkeerd met een `?`, echte constituenten met een `!`.

```
(21)  ( ! NPS )
      ( ? AUX )
```

Het blijkt mogelijk verwerkte stukjes invoer in de invoerstring telkens te vervangen door het label van de herkende constituent. Gebruik maken van de invoerstring voor het bewaren van herkende constituenten heeft twee grote voordelen. Ten eerste zijn geen extra parameters in het `parse`-programma nodig. Ten tweede wordt de volgorde van de invoerstring automatisch bewaard.

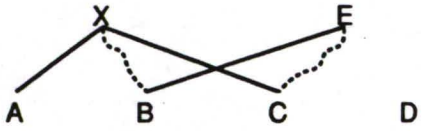
Er is echter ook een nadeel: De invoerstring moet aan het eind van een succesvolle parse leeg zijn, omdat we dan de zekerheid hebben dat alle stukjes invoer ook daadwerkelijk gebruikt zijn bij het herkennen van de structuur. Deze stopconditie willen we graag handhaven. En dat kan.

Er zijn herkende constituenten die niet meer in volgende regels een rol kunnen spelen. Dat zijn alle echte constituenten die voorafgaan aan de eerste reeds herkende contextconstituent.

Waarom is dat zo? Dat is zo omdat

1. herkende echte constituenten alleen nog als interne contextconstituent gebruikt kunnen worden in volgende regels, en
2. interne contextconstituenten altijd tussen een echte linker en een echte rechter constituent moeten staan.

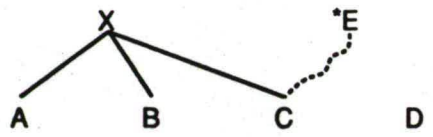
Echte constituenten waaraan geen contextconstituent voorafgaat kunnen niet meer elders als contextconstituent voorkomen omdat geen constituent in aanmerking komt als echte linker constituent. Enkele voorbeelden zullen dit duidelijk maken:



Figuur 12: C is contextconstituent onder E

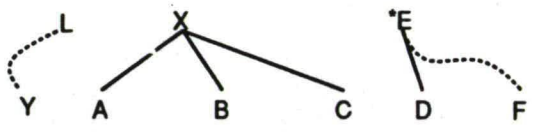
Stel dat E een rechterbuur moet zijn van X. In figuur 12 kan C onder E contextconstituent zijn omdat B onder E als echte dochter kan worden gebruikt.

In figuur 13 kan C onder E géén contextconstituent zijn omdat geen echte linker buur voor C onder E beschikbaar is (B is geen context meer onder X).



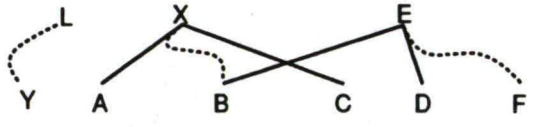
Figuur 13: C kàn geen contextconstituent onder E zijn

Het is belangrijk in te zien dat contextelementen van structuren links van X niet beschouwd hoeven te worden: die context elementen gaan in de precedentierelatie vooraf aan X. Alle echte constituenten van E links van de meest linkse contextconstituent van E hebben dezelfde plaats in deze ordening als E, dat wil zeggen ze volgen op X. Contextelementen uit L die gebruikt worden als echte dochter in E zouden zo zowel voorafgaan aan als volgen op X (figuur 14). Y kan wèl dochter zijn van een topstructuur van L, X of E. Stel dat L een linkerbuur moet zijn van X :



Figuur 14: L moet een linkerbuur zijn van X

Dit is een locaal criterium. Dat wil zeggen dat de beslissing of echte constituenten van X elders als contextconstituent kunnen voorkomen alleen afhangt van informatie die beschikbaar is uit constituenten die door X gedomineerd worden (de scope van X). Zo mag een *contextelement* van B weer wél onder E gebruikt worden:



Figuur 15: Contextelement van B gebruikt onder E

Het subprogramma `strip` zorgt ervoor dat de 'invoerstring' uiteindelijk leeg kan zijn door echte dochters links van de meest linkse contextconstituent uit de invoerstring te verwijderen en werkt als volgt (22, 23):

```
(22) ((strip ((! _c)|_rest) _result)
        (strip _rest _result))

(23) ((strip _x _x ))
```

(22) Betekent: als het eerste element uit de invoerparameter een echte dochter is verwijder dan dat element uit de invoerlijst. De `_rest` van die lijst moet verder

gestript worden: er kunnen immers nog meer echte dochters in de lijst staan, links van de eerste contextdochter of van het eerste nog niet herkende element in de lijst.

De tweede clause van strip (23) verandert expliciet niets aan de invoerparameter: de uitvoerparameter wordt door unificatie gelijk gemaakt aan de invoerparameter. Als het eerste element van de invoerparameter òf een nog niet geparseerd terminaal symbool is, òf een als interne contextconstituent herkend symbool dan moeten alle elementen in de invoerlijst bewaard blijven. Terminale symbolen zijn nog niet herkend, contextelementen moeten nog een echte moeder krijgen, en echte dochters die mogelijk voorkomen na een contextelement kunnen nog als interne context nodig zijn voor later te bouwen constituenten.

Hoe volgt nu dat ten laatste de invoerlijst leeg is? Als het `parse` proces klaar is, en er staan nog niet herkende elementen in de lijst, of contextelementen, dan is niet aan de hele zin structuur toegekend, en behoort de zin niet tot de taal. Echte dochters staan niet in de lijst, want de constructie van de topstructuur roept immers ook `strip` aan: zij 'weet' immers niet dat ze topstructuur is, en verwijdert alle elementen uit de lijst die gemarkeerd zijn als 'echte dochter'.

Hieronder volgt de parser voor discontinue structuren (24 t/m 28).

```
(24) ((parse () _x _x () ))

(25) ((parse (([_c])|_cs) ((_s _c)|_r1) ((_s _c)|_rest)
                                          (([_c])|_str-sis) )
        (strip _r1 _r2)
        (parse _cs _r2 _rest _str-sis))

(26) ((parse (([_c])|_cs) _words ((? _c)|_rest)
                                 (([_c])|_str-do)|_str-sis) )
        (rule _c _dochters)
        (parse _dochters _words _words1 _str-do )
        (strip _words1 _words2)
        (parse _cs _words2 _rest _str-sis))

(27) ((parse (_cat|_cats) ((? _c)|_r1) ((! _c)|_rest)
                                          ((_c)|_strsis) )
        (strip _r1 _r2)
        (parse _cats _r2 _rest _strsis))

(28) ((parse (_cat|_cats) _words ((!_cat)|_rest)
                                   ((_cat|_s-do)|_s-sis) )
        (rule _cat _dochters)
        (parse _dochters _words _r1 _s-do)
        (strip _r1 _r2)
        (parse _cats _r1 _rest _s-sis))
```

Aan de stopconditie (24) is niets veranderd: als de lijst met te herkennen constituenten leeg is, dan is op dat punt van de executie het proces klaar: de uitvoerparameter wordt geunificeerd met de invoerparameter. Als de invoerlijst leeg is, dan is ook het hele proces ten einde. De resultaatlijst is dan ook leeg, en komt dan overeen met de geeiste lege lijst uit de query. Verder hoeft er niets te gebeuren.

Regel (25) en (26) hebben betrekking op contextelementen. Regel (25) is er voor contextconstituenten die al eerder geparseerd waren en regel (26) voor nieuw te maken structuren.

Wanneer een contextconstituent (`[_cat]`) gevraagd wordt, en vooraan in de lijst met de invoerstring staat een lijst van de vorm `(_symbool _cat)` (in (25): `((_s _c)|_rl)`), dan is de gevraagde constituent al eens eerder herkend. Als `_symbool` een `!` is, gaat het om een echte dochter van een eerder geparseerde constituent, als het een `?` is, is het een contextdochter geweest. Voordat de rechterzusters `_cs` worden geanalyseerd moet elke echte dochter voor de eerste contextconstituent uit die lijst worden verwijderd. Daar zorgt `strip` voor. Vervolgens wordt `parse` weer aangeroepen om de rest van de categorieen uit de eerste parameter te herkennen.

Als de invoerstring als eerste element niet een al eerder herkende constituent heeft (26), dan moet er in de database naar een toepasbare regel worden gezocht. Als het `parse`-proces lukt wordt in de uitvoerparameter vóór de nieuwe rest `_rest` de lijst `(? _c)` gezet.

Deze twee clauses (25 en 26) zorgen samen voor het herkennen van contextconstituenten. De clauses onder (27 en 28) beschrijven wat te doen voor echte constituenten.

In (27) wordt een constituent `_c` gevraagd, en de invoerlijst bevat als eerste element een eerder als contextconstituent herkend element van de juiste categorie `_c`. In de invoerparameter `((? _c)|_rest)` wordt nu een echte dochter geplaatst die de eerdere contextdochter vervangt. Vervolgens wordt geprobeerd de rechterdochters `_cats` van de huidige constituent te parseren. Als de andere `_cats` uit de kop van de clause herkend kunnen worden wordt het `?` vervangen door een `!`: de contextconstituent is als echte dochter herkend.

(28) Behandelt het meest algemene geval: haal een regel op, pas die toe, verwijder wat mogelijk is uit de 'invoerstring', en ga verder met de rest van de constituenten uit de lijst.

4. De compilers voor DPSG-grammatica's

Zoals reeds eerder is opgemerkt kunnen de bestanden met het lexicon en met de regels niet zonder meer aan een parser worden aangeboden: de vorm waarin zij zijn opgeslagen is meer geschikt voor de menselijke gebruiker dan voor automatische verwerking door de parser. De bestanden moeten eerst bewerkt worden door een ander programma dat ze aanpast aan de behoeftes van die parser. Zo een programma heet een compiler. Deze controleert of de bestanden voldoen aan de formele definitie, en slaat de informatie die in de bestanden vervat is op in een datastruktuur die die geschikt is voor de parser.

In dit hoofdstuk beschrijf ik de datastructuur waarin de informatie zal worden opgeslagen. Voorts laat ik zien hoe de BNF definitie van de invoerbestanden in Prolog geoperationaliseerd is.

4.1. De representatie van het DPSG-formalisme in Prolog

Voor elke regel en elk entry in het lexicon wordt een Prolog feit aan de database toegevoegd. In sommige gevallen meer: wanneer een lemma meerdere interpretaties heeft wordt voor elk van die interpretaties een feit toegevoegd, en voor regels met optionele constituenten worden net zoveel feiten aan de database toegevoegd als nodig is.

Elk feit bestaat uit een Prolog predikaatnaam, gevolgd door de informatie uit de regel of uit het lexicon-entry. De informatie uit de regel of het lexicon-entry wordt per regel of entry opgeslagen in telkens een lijst. Voor zowel regels als entries bevat deze lijst dezelfde onderdelen, omdat regels en items dezelfde soort informatie bevatten.

De regels bevatten zeven soorten informatie, waarvan er drie net zo voorkomen bij de lexicon-entries. De vier soorten informatie die bij de entries ontbreken hebben betrekking op de voorwaarden waaronder een regel toegepast kan worden. Het zijn condities op kenmerken van de delen die worden gecombineerd. De informatie over condities mag bij lexicon-items ontbreken omdat een lexicon-entry geen zinsdelen combineert.

In de lijst die de informatie over een lemma bevat wordt voor elk van de vier soorten ontbrekende informatie een lege component toegevoegd. Dit betekent dat bij lemmata telkens vier lege lijsten voorkomen als onderdeel van de lijst met informatie. Door voor regels en entries dezelfde datastructuur te kiezen wordt het parseralgoritme redelijk eenvoudiger. Zie daarvoor Paragraaf 3.2.

Hieronder treft u een voorbeeld aan van een lexicon-entry (1) en het bijbehorende Prolog feit (2)

```
(1) HET      \2      CENTRALDET
                     FORM SING GENDER NEUTR *
                     constant CRo ;
```

```
                    CENTRALDET
                    FORM MASS GENDER NEUTR *
                    constant CRo ;

(2)  ((regel centraldet (het)
                        (
                          () (/* locale attribuutcondities)
                          () (/* globale attribuutcondities)
                          () (/* carry over of attribute values)
                          () (/* incidental carry over)
                          ((form sing) (gender neutr))
                        )
                        (constant cro)
     ))
```

Voor het woord HET wordt er door de compiler nog zo een feit in de database gezet, met als enige verschil dat er daar op de plaats van de waarde `sing` voor het `form`-attribuut de waarde `mass` staat (de tweede interpretatie van HET, die in (1) na de ; begint).

In de Prolog-database is het verschil tussen een lexicon-entry en een regel dat de lege lijsten op de plaats voor de locale en globale attribuutcondities en de lijsten voor globale en incidentele kenmerkoverdracht daar gevuld zijn met de desbetreffende informatie uit de regel.

4.2. Het operationaliseren van de BNF-definities in Prolog

De compilers hebben, zoals opgemerkt, een tweeledige functie. Enerzijds moeten zij de invoerbestanden op formele correctheid controleren, anderzijds dragen zij zorg voor de vertaling naar de datastructuur.

Het controleren van de regels en het lexicon. De controle wordt gerealiseerd door de BNF-definities operationeel te maken. Daartoe zijn zij in Prolog geimplementeerd. Door de keuze voor Prolog als programmeertaal was de implementatie van de controlefase relatief eenvoudig. Ter illustratie volgt hieronder de BNF-regel voor een 'valexpr' (3), met het daarbij behorende pseudo Prolog programma (4).

```
(3) valexpr ::= attval | '[' attvallst | '{' attvalset

(4) (valexpr) als (attval)
    (valexpr) als ('[') en als (attvallst)
    (valexpr) als ('{') en als (attvalset)
```

In het programma is de disjunctie vertaald in één Prolog regel voor elk van de drie alternatieven. Wanneer nu de invoer niet voldoet aan de eisen die in een van de drie regels daaraan worden gesteld zal de invoer worden afgekeurd.

De manier waarop Prolog de invoerstroom behandelt leidt nog tot een tweetal observaties. Neem bijvoorbeeld de volgende BNF-regel, die uitdrukt dat A herschrijft als a gevolgd door Q of als z:

```
(5) A ::= a Q | z
```

Dit wordt in (pseudo) Prolog uitgedrukt als:

```
(6) (A) als (a) en als (Q)
    (A) als (z)
```

Neem ook de volgende stand van zaken aan in het invoerbestand (7a): Wanneer op

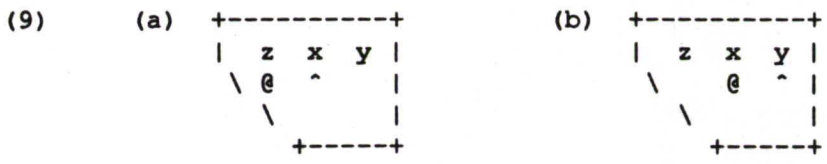
```
(7)      (a)  +----------+          (b)  +----------+
              |   z  x   |               |   z  x   |
               \  ^       |               \      ^   |
                 \        |                 \        |
                   +-----+                   +-----+
```

dit moment het (sub)programma A wordt aangeroepen zal de eerste A-clause het eerst worden geprobeerd. Die probeert een a te lezen, maar treft een z aan. Bij het lezen van die z wordt tevens de filepointer ^ verplaatst naar de x (zie 7b). Omdat de gelezen letter z niet overeenkomt met de gevraagde stopt hier de executie van de eerste A-clause. Bindingen van variabelen worden ongedaan gemaakt waardoor de gelezen z verloren gaat. Het Prolog systeem probeert de tweede A-clause. Nu wordt een z gevraagd, maar bij het lezen wordt een x aangetroffen. Zo levert ook deze tweede A-clause geen succes op en failt het gehele A- programma.

Door gebruik te maken van een look-ahead variabele kan dit probleem worden ondervangen. De waarde van de look-ahead variabele is in dit geval het laatstgelezen symbool. Door dit symbool als parameter mee te geven gaat het niet meer verloren bij de keuze tussen verschillende alternatieven. We herschrijven daartoe het A-programma als volgt:

```
(8) (A 'a' _nxt) als (lees 'a' nxt1) en als
                     (lees nxt1 _nxt)
    (A 'z' _nxt) als (lees 'z' _nxt)
```

Hierin is de eerste parameter de invoer-, de tweede de uitvoerparameter. De eerste wordt ter plekke verwerkt, de tweede ter plekke gelezen, en wordt bewaard ten behoeve van een willekeurig volgend (sub)programma. Nu is de situatie in het invoerbestand (9a): in de aanroep van A staat nu op de plaats van de invoerparameter @ een z (afkomstig uit een eerder uitgevoerd (sub)programma); de eerste clause wordt nu overgeslagen, omdat immers niet wordt gematched. De tweede clause levert succes op, met bovendien _nxt (^) gebonden aan de x die in de invoerstroom werd aangetroffen.



In (9b) is z verwerkt, x staat als invoerparameter (@) klaar voor het volgende programma en de filepointer (^) staat op het teken volgend op de x. Door dus gebruik te maken van een look-ahead variabele wordt beperkt niet destructief backtracken mogelijk.

De andere observatie betreft het gebruik van de disjunctie operator in BNF in vergelijking met de (impliciete) disjunctie in Prolog. Het systeem van probleemoplossen door backtracken dat door Prolog wordt gebruikt strekt zich niet uit tot in- en uitvoeroperaties. Dat heeft gevolgen voor het gebruik van de keuze operator in de beschrijving van de bestanden. Vergelijk bijvoorbeeld de BNF-regel hieronder die uitdrukt dat K herschrijft als a gevolgd door b of als a gevolgd door c:

```
(10) K ::= a b | a c
```

Deze regel wordt in Prolog uitgedrukt met behulp van twee K-clauses. De eerste voor het herkennen van de sequentie a b, de tweede voor de reeks a c. Wanneer het Prolog programma de invoer gelezen heeft en herkend heeft als a wordt eerst een b verwacht. Bestaat de invoer nu niet uit b, dan probeert Prolog de tweede K-clause. Dit backtrack proces houdt in dat vorige instanties van variabelen ongedaan worden gemaakt. In de tweede K- clause wordt weer een a verwacht die echter niet opnieuw kan worden gelezen.

De propositielogische regel dat

```
(a & b) v (a & c) <==>  a & (b v c)
```

betekent dat K geschreven mag worden als

```
(11) K1 ::= a B
     B  ::= b | c
```

waarin K1 overeenkomt met a & (b v c) en B met (b v c). Het Prolog programma voor de regel K1 bestaat nu uit een test op de aanwezigheid van a, en een aanroep van B, zonder dat de a opnieuw nodig is. Dit illustreert hoe operaties met een neveneffect, verplaatsing van de filepointer, tot gevolg hebben dat Prolog programma's niet zuiver predikaatlogisch, zelfs niet zuiver propositielogisch zijn.

4.2.1. Het genereren van Prolog-representaties

Tijdens het lezen van de file worden per regel of lex-entry steeds afgeronde stukken informatie opgeslagen in variabelen. Wanneer alle informatie uit een regel of entry bekend is (voor een regel is dat wanneer het teken `-*-` is gelezen, voor een lexicon-entry wanneer een `;` is gelezen) wordt de informatie in de beschreven datastructuur gepast: er wordt een Prolog-fact aan de interne database toegevoegd. Ik beschrijf hier het mechanisme zoals het gebruikt wordt voor de regelfile; voor de lexiconfile geschiedt één en ander analoog.

Voor het verzamelen van de informatie uit de regels zijn drie hoofdprocedures van belang die in het programma `parts` worden aangeroepen (zie appendix B, pp 63 ff.):

- `Part-1` levert de structurele informatie:
  - de topknoop die door een regel wordt gegenereerd,
  - de dochters waaruit die knoop bestaat,
  - de selectoren die bij die dochters horen,
  - of er optionele constituenten in de regel voorkomen, met andere woorden: moet er meer dan een regel worden gegenereerd
- `Part-II` verzamelt alle informatie die betrekking heeft op de
  - condities op kenmerken en op
  - kenmerkoverdracht
- `Part-3` levert de semantische formule op die hoort bij de regel.

De werking van `part-II` bijvoorbeeld is als volgt:

```
(12) ((part-II _f _fp _nxt _attr-conds)
        (part2 _f _fp _nxt1 () _lc)
        (part3 _f _nxt1 _nxt2 _lc _lgc)
        (part4 _f _nxt2 _nxt3 (_lgc) _lgcr)
        (part5 _f _nxt3 _nxt4 _lgcr _lgcri)
        (part6 _f _nxt4 _nxt _lgcri _attr-conds))
```

De eerste variabele in de head (`_f`) bevat de naam van de file die gelezen wordt. `_fp` bevat het laatste teken dat in het aanroepende programma gelezen is en `_nxt` is op het moment van aanroep een vrije variabele. Die zal, als `part6` succesvol is geëxecuteerd, worden geïnstantieerd met het eerste token na het gedeelte in de regel dat de kenmerkinformatie bevat. De laatste parameter (`_attr-conds`) is op het moment van de aanroep ook een vrije variabele. Aan het eind van de executie van `part6` staat in die variabele alle informatie over de kenmerken.

De aanroepen van de parts 2 tot en met 6 bevatten voor het opsparen en doorspelen van de kenmerkinformatie steeds twee variabelen. In `part2` is de eerste van die variabelen geinstantieerd met de lege lijst: wanneer we beginnen met informatie verzamelen hebben we immers nog niets. In het programma `part2` worden in `_lc` de locale attribuutcondities opgeslagen. `_lc` is op zijn beurt weer inputvariabele voor `part3`. In `part3` worden de globale attribuutcondities aan `_lc` toegevoegd, met als resultaat `_lgc`. `part4` gebruikt `_lgc` weer als invoerparameter, en zo voorts, tot en met `part6`, waar de uitvoerparameter uiteindelijk de verzamelde informatie over de kenmerken bevat in `_attrconds`. Door unificatie is deze variabele in de head nu ook geïnstantieerd.

Wanneer het symbool `-*-` in de regelfile is bereikt is alle informatie die van belang is voor een regel in de een of andere vorm beschikbaar. Deze informatie moet nu in de vorm van een Prolog-feit aan de interne database van het Prolog systeem worden toegevoegd. De vorm van zo'n Prolog-feit is hierboven in 3.3. beschreven. Het Prolog-programma `maak-rules` giet de gegevens in de vereiste vorm.

In de meeste gevallen kan worden volstaan met een beperkte reorganisatie van de variabelen die als invoerparameters binnenkomen. Maar, zoals in 1.3. is opgemerkt, er kunnen regels voorkomen met optionele constituenten zoals in (13) hieronder:

```
(13) a:A + (b:B) + c:C --> r:X
```

(13) Is een combinatie van de twee regels (14)

```
(14) a:A + b:C --> r:X
     a:A + b:B + c:C --> r:X
```

Wanneer zo'n regel is gelezen, moet die geëxpandeerd worden: voor elk van de mogelijke regels moet een feit worden toegevoegd. Hoeveel dat er zijn wordt bepaald door de machtsverzameling optionele elementen. Voor iedere deelverzameling van de verzameling optionele elementen worden alle elementen uit de PC-regel verwijderd (`pc-variant` in `maak-rules`, appendix B). De lege deelverzameling levert zo de PC-regel waarin alle optionele elementen voorkomen. Daarbij moet telkens worden gecontroleerd of er verwijzingen binnen een regel plaatsvinden naar constituenten die er niet zijn. Voor een regel dienen condities waarin die verwijzingen voorkomen verwijderd te worden. `maak-rules` bepaalt het aantal feiten dat moet worden toegevoegd en draagt zorg voor de administratie.

Er is maar een doorgang door de bestanden nodig: de organisatie van de invoerbestanden is zodanig dat er nergens verwijzingen plaatsvinden tussen regels. Alle informatie in een entry of regel is locaal voor dat entry, voor die regel.

5. De parser voor DPSG-grammatica's

DPSG-regels zijn bedoeld voor het verwerken van natuurlijke talen. Zij bevatten, behalve informatie over het aantal constituenten in een phrase, hun volgorde en categorienamen ook nog condities, kenmerken en een semantische regel. In de compiler van hoofdstuk 3 is behandeld hoe deze informatie in Prolog wordt gerepresenteerd.

In dit hoofdstuk wordt de parser uit hoofdstuk twee aangepast en uitgebreid ten behoeve van deze uitgebreidere regels. Allereerst moet de parser aangepast worden aan de vorm van de DPSG-regels zoals die door de compiler uit hoofdstuk drie worden gegenereerd. Dit komt in paragraaf 4.1. aan de orde.

Ten tweede moeten de condities worden geevalueerd en moeten kenmerken worden toegekend. Deze evaluator wordt in 4.2. behandeld.

Tenslotte wordt met behulp van de semantische regel een semantische formule toegekend. Hoe dat gebeurt, staat in 4.3.

5.1. De aanpassingen van de parser aan de vorm van de DPSG-regel

In hoofdstuk 3 hebben we gezien dat een DPSG-regel in Prolog de vorm heeft

```
(1) ((regel _cat _dochters _attconds _semexpr))
```

De compiler uit hoofdstuk 3 representeert grammaticaregels en lexicale eenheden in deze vorm. `_cat` Bevat het label van de phrase structuur die met behulp van deze regel kan worden gemaakt. In `_dochters` staan de categorienamen van de constituenten van de phrase met categorienaam `_cat`.

Voor iedere constituent is er een eigen categorienaam. De volgorde van deze namen in de lijst is de volgorde waarin de constituenten voor deze regel in de structuur moeten voorkomen.

De variabele `_attconds` bevat de informatie omtrent de condities op de attributen van deze constituenten alsook welke van hun attribuutwaarden overgedragen moeten worden op de resulterende phrase-structuur.

`_semexpr` Bevat de semantische regel die vastlegt hoe de semantiek voor deze regel opgebouwd wordt uit die van de constituenten.

`_Attconds` En `_semexpr` kwamen niet voor in de regels voor de structuurparser uit hoofdstuk 2. Waar in de clausules van die parser een regelstructuur voorkomt moeten deze nieuwe variabelen toegevoegd worden.

In elke `parse`-clausule wordt telkens één constituent geanalyseerd. Dit betekent dat de vorm van die constituent in de betreffende clausule in overeenstemming moet worden gebracht met bovenstaande regelvorm. De lijst van constituenten had de vorm `(_cat | _cats_)`. De aanpassing bestaat erin dat twee procesvariabelen `_atts` en `_sem` worden toegevoegd, zodat de lijst `(_cat|_cats)` de vorm krijgt

```
(2) ( (_cat _atts _sem) | _cats)
```

De aanroep van `parse` wordt zo:

```
(3) ((parse (( _cat _atts _sem)|_cats) _invoer _uitvoer _struc))
```

Deze aanpassingen aan de structuurparser zijn voldoende om boomstructuren te kunnen opleveren met DPSG-categorienamen `_cat` als labels, met attribuutinformatie `_atts` en semantische representatie `_sem`. De lijst `(_cat _atts _sem)` representeert nu alle informatie van een knoop in de structuur.

Door het programma `eval` krijgen de procesvariabelen `_atts` en `_sem` hun waarde. In de volgende paragraaf laat ik zien hoe de informatie over attributen van de constituenten wordt gebruikt om enerzijds mogelijke structuren uit te sluiten met behulp van attribuutcondities en on anderzijds de waarde vast te stellen van de attributen van de resulterende structuur (carry-over en new values). Bovendien wordt in de evaluator de semantische expressie `_sem` opgebouwd die hoort bij de te bouwen phrase.

5.2. Toevoegingen t.b.v attribuutverwerking en semantiek

Er zijn twee parse-clauses die uit de Prolog database een regel ophalen. Voor alle constituenten uit deze regel wordt geprobeerd een structuur te parseren. Als dat gelukt is kunnen de attribuutcondities uit de regel worden geevalueerd (eerste paragraaf).

Als de constituenten aan alle condities voldoen kunnen waarden worden toegekend aan attributen van de phrase die met deze regel wordt geanalyseerd (tweede paragraaf). Tenslotte moet nog een semantische expressie worden samengesteld (derde paragraaf).

De twee parse-clauses (24) en (27) op p.*** krijgen hiertoe een extra subgoal: een extra aanroep van het programma `eval`. `eval` Draagt achtereenvolgens zorg voor de evaluatie van de condities op attributen van constituenten, voor het overdragen van kenmerken en attribuutwaardes naar het resultaat, en voor het construeren van de semantische expressie.

Na deze toevoegingen zien (24) en (27) er als volgt uit:

```
(24') ((parse ((_cat _atts _sem)|_cats) _wrds ((?_cat)|_rest) )
      i)      (rule ... )
      ii)     (parse ... )
      iii)    (eval ... )
      iv)     (strip ...)
      v)      (parse ...))
```

In `i)` wordt een regel genomen. In `ii)` wordt geprobeerd voor elk van de constituenten die in de regel worden genoemd een structuur te maken. In `iii)` worden de condities op deze constituenten geëvalueerd. In `iv)` worden de constituenten die niet meer als contextelement kunnen voorkomen verwijderd en in `v)` tenslotte worden de rechterburen van de structuur die met deze regel gemaakt is geparseerd.

5.2.1. De evaluatie van attribuutcondities

In paragraaf 1. hierboven hebben we gezien dat een DPSG-regel in Prolog de vorm heeft:

```
(4) ((regel _cat _dochters _attconds _semexpr))
```

De variabele `_attconds` bevat de attribuutcondities en kent vier elementen (zie hoofdstuk 3):

```
(5) (_globalcarry _incicarry _newvals _condities)
```

Elk van deze elementen heeft de vorm van een lijst. In deze paragraaf behandel ik alleen de vierde lijst, die alle attribuutcondities bevat. De eerste drie komen in de volgende subparagraaf aan bod.

De lijst `_condities` bevat attribuutcondities van de vorm

```
(6) (_selector1 _selector2 _attribuut)
```

of

```
(7) (_selector (_relatie _attribuut _waarde))
```

De eerste vorm duidt op een globale attribuutconditie, de tweede op een locale attribuutconditie. Het verschil is dat een locale conditie geen verband legt tussen attributen van zusterknopen, een globale conditie juist wel. De globale conditie

```
(8) (a c GENDER)
```

legt verband tussen de twee constituenten `a` en `b` door te eisen dat zij voor het attribuut `GENDER` dezelfde waarde hebben. De conditie

```
(9) (a (eq GENDER MASC))
```

is een locale conditie: zij geldt slechts voor de constituent `a`. De waarde van het attribuut `GENDER` voor die subknoop moet `MASC` zijn.

Het programma `eval-lcgc` controleert of de gevonden constituenten voldoen aan de eisen die in de condities worden gesteld. Het programma heeft twee parameters: de eerste is de lijst constituenten, de tweede is de lijst met condities op de attributen ervan.

`eval-lcgc` Bestaat uit drie clauses; de eerste clause is de stopconditie: als de tweede parameter de lege lijst is zijn alle condities verwerkt en is `eval-lcgc` klaar. De twee andere clauses behandelen het geval van de niet-lege lijst. Zolang de lijst niet leeg is is de werking afhankelijk van de vorm van de eerste conditie in de lijst: heeft die de vorm van een globale conditie, namelijk `(_s1 _s2 _attr)`, dan wordt het programma `check-gac` (check global attribute conditions) aangeroepen, anders `check-lac` (check local attribute conditions). De laatste subgoal van beide clauses is de recursieve aanroep. Daarbij wordt de lijst met attribuutcondities minus de eerste conditie als tweede parameter meegegeven (`_rest`):

```
(10) ((eval-lcgc _dochters ())/)
     ((eval-lcgc _dochters ((_s1 _s2 _attr)|_rest))
        (check-gac _dochters _s1 _s2 _attr)
        (eval-lcgc _dochters _rest))
     ((eval-lcgc _dochters ((_sel (_rel _attr _val))|_rest))
        (check-lac _dochters _sel (_rel _attr _val))
        (eval-lcgc _dochters _rest))
```

Het programma `check-gac` vergelijkt de waarde die aangetroffen is voor het attribuut `_attr` bij constituent `_s1` met de waarde die voor `_attr` is aangetroffen bij constituent `_s2`. Het programma failt als de waarden niet gelijk zijn.

`check-lac` Controleert of de relatie `_rel` opgaat tussen attribuut `_attr` en zijn waarde `_val` voor een bepaalde constituent (`_sel`). Dit programma failt als de relatie niet opgaat.

5.2.2. Kenmerk-overdracht

We hebben in de subparagraaf hierboven herhaald dat de lijst van attribuutcondities in de Prolog representatie van de DPSG-regels bestaat uit de vier lijsten

```
(11) (_globalcarry _incicarry _newvals _condities)
```

De verwerking van de condities is hierboven uiteengezet. Voor de verwerking van de eerste drie lijsten zorgen de twee programma's `eval-gc` (locale overdracht) en `eval-ic` (incidentele overdracht), samen met het programma `update` (nieuwe waarden). Deze drie programma's dragen zorg voor de overdracht van kenmerken van de subknopen naar de resultaatphrase.

De lijst `_globalcarry` bevat een enkele selector: alle attribuutwaarden die horen bij die subconstituent gaan over naar de resulterende structuur. Het programma `eval-gc` verzamelt alle attribuut-waarde paren.

Het programma `eval-ic` haalt selectief dergelijke paren uit de verzameling kenmerkinformatie van een subconstituent.

In `_newvals` zijn nieuw toe te kennen attribuutwaarden opgenomen. Beide uitvoerparameters, samen met de informatie uit `_newvals`, gaan naar het programma `update`, waar de lijst attribuut-waarde paren voor de resultaatknoop wordt opgebouwd. Deze lijst bestaat weer uit lijsten van de vorm

```
(12) (_attribuut _waarde)
```

De volgorde van toekenning is als volgt: eerst worden alle kenmerken uit de lijst `_globalcarry` toegevoegd. Vervolgens wordt deze lijst met attribuut-waarde paren eventueel aangepast of uitgebreid aan de hand van de gegevens uit de lijst `_incicarry`. Tenslotte worden de `_newvals` toegekend.

In de eerste paragraaf hierboven hebben we gezien dat de heads van de parse-clauses voor DPSG de vorm hebben:

```
(13) ((parse (( ([_cat]) _atts _sem)|_rest) _invoer _uitvoer _struc))
     ((parse ((   _cat   _atts _sem)|_rest) _invoer _uitvoer _struc))
```

waarbij `_atts` en `_sem` op het moment van de aanroep ongebonden variabelen zijn. Nadat de condities op de toepasbaarheid van de regel zijn geevalueerd, en een lijst met attribuut-waarde paren voor de resultaatknoop is opgebouwd wordt `_atts` middels unificatie gebonden aan het resultaat van `update`, de lijst attribuut-waarde paren van de resultaatknoop. Indien de evaluatie van de attribuutcondities, of het toekennen van waardes aan de gebouwde phrase niet lukt wordt de hele phrase als niet gebouwd beschouwd, en backtrackt het Prolog systeem zodat een nieuwe (nog niet eerder toegepaste) regel wordt geprobeerd.

5.2.3. Het opbouwen van een semantische expressie

De laatste lijst van een Prolog-DPSG regel bevat de semantische regel die bij de phrase hoort. In die regel kan de subexpressie

```
(14) (parameter _sel)
```

voorkomen. Die subexpressie wordt in het programma `eval-sem` vervangen door de semantische expressie die hoort bij constituent `_sel` van de phrase. Aan het eind van een succesvolle parse bevat de ongebonden variabele `_sem` uit de call zo een ongereduceerde semantische expressie die de betekenis van de phrase uitdrukt in EL/f.

6. Het coderen en testen van de software

e software die in deze scriptie is gepresenteerd is in de loop van de ontwikkelcyclus op tal van aspecten gecontroleerd en in de praktijk beproefd. In het volgende beschrijf ik kort een aantal aspecten van het coderen en controleren van de software.

6.1. Het coderen en testen van de compiler

De implementatie van de compiler is geschied in twee fases. Eerst is de controlefase gecodeerd; pas toen de compiler in staat was zonder problemen een compleet regelbestand te lezen is de 'verzamelfase' gecodeerd.

Het coderen van het lezen is een directe omzetting van de BNF-definities in de appendix naar Prolog. Hoe die omzeting in zijn werk gaat is al beschreven in 1.2.4 en in 3.2. Ik ga er hier niet verder op in.

Het coderen van de 'verzamelfase' was een vertaling van de informatie die in de regels is opgenomen naar de datastructuur voor de parser. De hoofdlijnen van dat proces staan beschreven in 3.2.1. Eén aspect dient hier echter apart vermeld te worden. Dat betreft de behandeling van de attribuutinformatie in de DPSGF-regels.

Tijdens het coderen van de verzamelfase is de beslissing genomen om alleen attribuut-waarde paren in de gegenereerde regels te plaatsen die in de DPSG-regels worden genoemd. Dit werd een probleem toen tijdens het testen van de DPSG parser sommige attribuutwarden niet werden herkend.

Het gaat met name om de waarde NOVAL. Deze attribuutwaarde wordt in het TENDUM systeem als verstekwaarde gegenereerd door de Pascal regel-en lexiconcompilers. In de Prolog regel- en lexiconcompiler wordt geen rekening gehouden met deze verstekwaarde, doordat die alleen datgene in de Prolog-regels opneemt wat daadwerkelijk in de DPSG-regels staat. Daardoor failt de parser wanneer in een regel een conditie op zo'n default waarde is opgenomen: in de regel wordt dan namelijk een conditie gelegd op een attribuut dat er in de Prolog representatie van de regel niet is.

De compiler is tijdens het ontwikkelen getest met een representatieve set regels uit de bestanden GRAM.RUL en NEDELF.LEX. Pas nadat gebleken was dat het lezen van de files probleemloos verliep zijn de twee bovengenoemde bestanden in hun geheel tot Prolog-regels gecompileerd. Daarbij kwamen geen noemenswaardige fouten aan het licht.

Het programma is bedoeld voor gebruik op een PC en niet geschikt voor het compileren van grote grammaticabestanden. Ten eerste is de implementatietaal niet

erg snel -het is een geïnterpreteerde taal- en daardoor neemt het compileren van grote aantallen regels veel tijd in beslag. Bij 'veel regels' dient u te denken aan 'meer dan dertig'.

Ten tweede is de geheugenbelasting erg groot. Dit betekent dat op een PC met 256Kb geheugen niet meer dan 100 regels kunnen worden gecompileerd. Dit zou kunnen worden verbeterd door in plaats van in het geheugen te compileren elke gegenereerde regel onmiddelijk naar een bestand te schrijven. Dat vertraagt de operatie weliswaar, omdat schrijven naar het externe geheugen immers veel meer tijd vergt dan schrijven naar het werkgeheugen, maar maakt het wel mogelijk ongeveer 3000 gecompileerde Prolog-regels te maken wanneer gebruik wordt gemaakt van een 360Kb diskette.

6.2. Het testen van de parser

6.2.1. Abstracte structuren

De structuurparser uit hoofdstuk 2.2 is uitvoerig gebruikt voor het analyseren van abstracte structuren waarvan de eigenschappen bekend waren. Een aantal voorbeelden treft u aan in appendix D. Deze voorbeelden spreken voor zich.

6.2.2. Nederlandse structuren zonder attributen

Op basis van de regels die door de compiler waren gegenereerd voor een subset van het Nederlands is een begin gemaakt met het testen van de DPSG-parser. In eerste instantie werden regels gebruikt zonder attribuutcondities of kenmerkoverdracht (ontleend aan [Aarts e.a. 1987]). Zoals te verwachten was werden veel meer zinnen goedgekeurd door de parser dan door een spreker van het Nederlands. De structuren voldeden echter wel aan de definities voor discontinue bomen zoals gegeven in 1.3.2. In appendix E treft u een aantal testzinnen aan met de gegenereerde structuur.

6.2.3. Nederlandse zinnen

De regels uit 5.2.2. hierboven werden uitgebreid met de attribuutcondities en regels voor kenmerkoverdract uit [Aarts e.a. 1987], en de zinnen uit 5.2.1. werden opnieuw aan de parser aangeboden. Ook dit verliep redelijk soepel; ook deze resultaten treft u aan in appendix F. De regels die zijn gebruikt treft U aan in appendix G.

Vervolgens is geprobeerd op basis van de grammatica in `GRAM.RUL` met `NEDELF.LEX` als lexicon weer andere zinnen te herkennen. Door de omvang van de grammatica duurde het parseerproces onredelijk lang (tot meer dan een uur voor de zin `De KL402 komt uit Montreal`). Bovendien werd geen structuur opgeleverd als ergens in de regels de condities op de default attribuutwaarde `NOVAL` werd genoemd. Er is verder van afgezien met deze grote bestanden verder te werken.

7. Slotwoord

n deze laatste pagina's beschrijf ik nog een paar tekortkomingen van de ontwikkelde software, en doe ik enige suggesties voor verbeteringen en aanpassingen. Eerst behandel ik de parser, vervolgens de compiler.

De uitvoer van de parser bestaat uit een tweetal lijsten:

- De lijst met alle attribuut-waarde paren die horen bij de gebouwde structuur
- De lijst met de ongereduceerde EL/f representatie behorende bij de gebouwde structuur.

Deze lijsten kunnen als invoer dienen voor modules ten behoeve van de pragmatische en semantische analyse van zinnen in een dialoogsysteem als TENDUM.

De lijst met attribuut-waarde paren bevat ook informatie die niet meer nodig is voor verdere analyse. Het betreft informatie van syntactische aard, zoals (GENDER FEMASC) en (FORM MASS). Dit in tegenstelling tot informatie uit attributen die betrekking hebben op pragmatische aspecten van de geanalyseerde zin zoals (MOOD INTERR) bijvoorbeeld. Waar de scheiding ligt tussen die soorten attributen is niet duidelijk. Het is ook niet de bedoeling dat een parser deze scheiding aanbrengt. Hier ligt echter wel een mogelijkheid om de uitvoer van de parser te optimaliseren.

Een tweede mogelijkheid ligt op het gebied van de semantische expressie. De semantische representatie die door de parser wordt opgeleverd is een *ongereduceerde* EL/f expressie. De parser zou kunnen worden uitgebreid met een mechanisme voor het reduceren van de expressies, hoewel ook dit strikt genomen niet tot de taken van een parser behoort.

Wellicht is voorts een koppeling van deze parser mogelijk met een uitgebreide versie van de software die door Hilde van der Togt [Van der Togt, 1988] in het kader van haar afstudeerwerk is ontwikkeld. Deze software gebruikt als invoer een gereduceerde EL/f expressie, en produceert op basis daarvan bedelingen uit het discussiemodel.

De parser die hier is ontwikkeld is, door de gekozen strategie, langzaam wanneer de regel die succes oplevert voor een bepaalde phrase niet als eerste regel in de database voorkomt. Dat wordt erger naarmate de gebruikte grammatica uitgebreider is. Omdat de parser de regels probeert toe te passen in de volgorde waarin ze in de Prolog-database staan kan het heel goed voorkomen dat, vóórdat de juiste analyse is gevonden, structuren worden opgebouwd, weer vergeten, en opnieuw gebouwd.

Een gedeeltelijke oplossing voor dat probleem is om gebouwde deelstructuren in één of andere datastructuur te bewaren, en tijdens het analyseproces telkens eerst die datastructuur te raadplegen of datgene wat gevraagd is wellicht al eens eerder gebouwd is. Zo een oplossing wordt onder andere in de Pascal-parser van het TENDUM systeem toegepast.

De parser kan niet omgaan met links-recursieve regels: die kunnen leiden tot een eindloze lus. Er zijn drie manieren om dat probleem op te lossen.

- De eerste vereist een grote mate van discipline bij de schrijver van een grammatica en behelst een verbod op het gebruik van links-recursieve regels.
- De tweede manier is door de compiler voor de regels zo aan te passen dat links-recursie herkend en gesignaleerd wordt. Deze eerste en tweede oplossing kunnen eventueel in combinatie worden gebruikt.
- De derde manier om problemen met links- recursie te voorkomen is door een andere parseerstrategie te gebruiken. De problemen met links-recursie zijn namelijk een rechtstreeks gevolg van de toegepaste top down, depth first, links naar rechts strategie.

Een volgend onderdeel van de parser dat voor verandering in aanmerking komt is het gedeelte voor de verwerking van attribuutcondities. De huidige Prolog-implementatie heeft als voordeel dat alleen die attributen gerepresenteerd worden die ook daadwerkelijk een waarde krijgen in de loop van het parse-proces.
De reden dat bij het testen van de parser met het grote GRAM.RUL bestand geen bevredigende resultaten konden worden geboekt is een aantal default aannames ten aanzien van de initialisatie van de attribuutwaardes. Deze default-kennis is nergens gerepresenteerd en maakt in de TENDUM parser deel uit van de software zelf (sic).

Eén van de mogelijke oplossingen zou zijn het initialiseren van een geordende lijst met *alle* mogelijke attributen in de compilatiefase (sjabloon ?), en vervolgens tijdens het parseren de attribuutcondities en attribuutwaardeoverdracht op die geordende lijst te doen plaatsvinden.

Tijdens het werken aan de compiler en parser zijn wijzigingen aangebracht aan de representatie van attribuutwaarden in het DPSG-formalisme. Met name betreft het hier het vervangen van atomaire waarden voor een aantal attributen door lijsten van waarden. Daarmee is in deze compiler en parser geen rekening gehouden.
Om deze nieuwe vorm van attribuutwaarden correct te kunnen verwerken moet de compiler worden uitgebreid, maar ook de evaluator `eval` van de condities op attribuutwaarden uit de parser zullen moeten worden aangepast.

Mogelijke uitbreidingen en verbeteringen van de hier ontwikkelde software zijn als volgt samen te vatten:

De compiler

- de behandeling en representatie van default attribuutwaardes
- de behandeling en verwerking van de nieuwe vorm van een aantal attribuutwaardes
- de behandeling van links recursieve regels

De parser

- het toepassen van een efficiënter algoritme
- het aanpassen van de evaluator aan de nieuwe vorm van een aantal attribuutwaardes
- het tonen op het scherm van een grafische weergave van de gebouwde structuur

A. De BNF-definities van de regel- en lexiconfiles

```
*** ALGEMEEN

regelfile           ::= [ comment ]  regelseq
regelseq            ::= { regel } 'end-of-rules'
regel               ::= regelnaam parts '-*-'
regelnaam           ::= { char } [ comment ]
comment             ::= '{' dump '}'
                      | '(*' dump '*)'
dump                ::= { atoom }
atoom               ::= ASCII(32) .. ASCII(126) { ASCII(32) .. ASCII(126) }
parts               ::= part1 part2 part3 part4 part5 part6 part7
partlabel           ::= dump ':'
selct               ::= 'a' .. 'z'  /'r'/
CAT                 ::= { 'A' .. 'Z' }
```

```
*** PART 1: Phrase structure rule

part1               ::= partlabel constit-sum resultaat [ constr-cl ] ';'
constit-sum         ::= constit rest-van-const-sum
rest-van-const-sum  ::= '+' constit-sum | '-->'
resultaat           ::= 'r' ':' CAT          \* normalconst met selct='r' *\
constr-cl           ::= '['  dump  ']'
constit             ::= const        | optconst
const               ::= contextconst | normalconst
optconst            ::= '(' const ')'
contextconst        ::= '[' normalconst ']'
normalconst         ::= selct ':' CAT
```

```
*** PART 2

part2               ::= partlabel [ attr-cnd-lst-loc ] ';'
attr-cnd-lst-loc    ::= { attr-cond }
```

```
*** GEZAMENLIJK VOOR PARTS 2, 3, 6

const-attr          ::= feat-sel attr-naam
feat-sel            ::= selct '.'
attr-naam           ::= atoom
value-expr          ::= attval | '[' attvallst | '{' attvalset
attvallst           ::= ']' | attval  rest-of-attvallst
attvalset           ::= attval rest-of-attvalset
rest-of-attvallst   ::= ',' attval rest-of-attvallst | ']'
rest-of-attvalset   ::= ',' attval rest-of-attvalset | '}'
attval              ::= atoom | '"' dump '"'
```

```
*** GEZAMENLIJK VOOR PARTS 2, 3

attr-cond          ::= const-attr rel value-expr
rel                ::= 'IN'           | 'in'           |
                       'NOT IN'       | 'not in'       |
                       'CONTAINS'     | 'contains'     |
                       'NOT CONTAINS' | 'not contains' |
                       '='            |
                       '>'
```

```
*** PART 3

part3              ::= partlabel [ attr-cnd-lst-glob ] ';'
attr-cnd-lst-glob  ::= { glob-attr-cond }
glob-attr-cond     ::= 'PERM' const-attr '=' '"' atoom '"'
                       | const-attr '=' const-attr
```

```
*** PART 4 \* Global Carry Over *\

part4              ::= partlabel [ carry ] ';'
carry              ::= selct '-->' 'r'
```

```
*** PART 5 \* Incidental Carry Over *\

part5              ::= partlabel [ incicarry ] ';'
incicarry          ::= incicarry-sum const-attr
incicarry-sum      ::= const-attr rest-van-carry-sum
rest-van-carry-sum ::= '+' incicarry-sum |
                         '-->'
```

```
*** PART 6: New attribute values

part6              ::= partlabel [ attr-cnd-lst-new ]
attr-cnd-lst-new   ::= { nw-attr-cnd }
nw-attr-cnd        ::= const-attr '=' value-expr
```

```
*** PART 7: Semantic rule
```

```
part7              ::= partlabel aandelen-lst formule ';'
aandelen-lst       ::= aandeel rest-van-aandelen
aandeel            ::= selct '''
rest-van-aandelen  ::= '+' aandelen-lst  |
                       '-->'
formule            ::= EL/f formule \* Zie definities in Bunt, 1985 *\
```

```
*** EXTRA voor LEXICON

lexfile            ::= [ comment ] wordseq
wordseq            ::= { word } "end-of-lex"
word               ::= woord nofvariants syncat attvals "*" semexpr ";"
woord              ::= atoom
nofvariants        ::= [ "/" digit ]
syncat             ::= CAT
attvals            ::= { attr-naam value-expr } |
                       [ attr-naam value-expr ]
semexpr            ::= EL/f formule
```

B. De Compiler voor de DPSG regel- en lexiconfiles

```
P "^L^B*** LOADING^D DPSG RULE & LEX COMPILERS^M^J."
?((KILL ALL))
((lees lex)/(compileer lexicon "nedelf.lex" ))
((lees rul)/(compileer rules "gram.rul"))

((e .)
  (unassign)
  (dos "edix lrcomp.log")
  (LOAD "f:assign")
  (reconsult lrcomp))
((e 1)
  (unassign)
  (dos "edix lrcomp.log")
  (LOAD "f:assign")
  (reconsult 1 lrcomp))

((compileer lexicon _f )
     / (OPEN _f)
     (interm _f _x)
     (lexfile _f _x)
     / (CLOSE _f))

((compileer rules _f )
     / (OPEN _f)
     (interm _f _nxt1)
     (regelfile _f _nxt1)
     / (CLOSE _f))
P .
((comment _f { _nxt) (interm _f _nxt1)
     (dump _f _nxt1  "}") (interm _f _nxt) )
((comment _f "(*" _nxt) (interm _f _nxt1)
     (dump _f _nxt1 "*)") (interm _f _nxt) )
((comment _f _fp _fp))

((partlabel _f _fp _nxt)
     (interm _f _nxt1) (dump _f _nxt1 ":")
     (interm _f _nxt) )

((eor _fp)
     / (EQ _fp "-*-") )
P .
((regelfile _f _fp)
     (comment _f _fp _nxt)
     (regelseq _f _nxt "end-of-rules")
     (P "*** ^Bregelfile " _f "^D correctly processed^M^J"))
((lexfile _f _fp)
     (comment _f _fp _nxt)
     (wrdseq _f _nxt "end-of-lex")
```

```
     (P "*** ^Blexiconfile " _f "^D correctly processed^M^J"))

((regelseq _f "end-of-rules"  "end-of-rules" ))
((regelseq _f _fp _nxt)
     (regel _f _fp _nxt1) /
     (regelseq _f _nxt1 _nxt) )

((dekken)
     / (KILL (aandelen gebonden-vars) )
     (ADDCL ((aandelen () )) )
     (ADDCL ((gebonden-vars () )) ) )
P .
((regel _f _fp _nxt)
     (dekken)
     (regelnaam _f _fp _nxt1 _naam)
     (parts _f _nxt1 "-*-" _sl _opt-elts (_kop|_dochters) _attrs _sem)
     (reduceer _opt-elts _opt-elts1)
     (maak-rules _opt-elts1 _kop _dochters _attrs _sem)
     (P "^Brule " _naam "^D correctly processed^M^J^M^J")
     (interm _f _nxt) )

((regelnaam _f _naam _nxt _naam)
     / (interm _f _nxt1)
     (comment _f _nxt1 _nxt) /)

((parts _f _fp _nxt _sellst _opts _catlst _attrs _semantiek)/
     (part1 _f _fp _nxt1 _sellst _opts _catlst)
     (part-II _f _nxt1 _nxt2 _attrs)
     (part7 _f _nxt2 _nxt  _semantiek) /)

((selct _a _sel)
    (ON _a (a b c d e f g h i j r))
    / (EQ _a _sel) )
((selct _a _a)
    (P "*** ERROR: not a valid selector:^B" _a "^D^M^J"))

((interm _file _fileptr)
     (INTOK _file _fileptr1)
     (EQ _fileptr1 _fileptr) /)
((interm _file "end-of-rules" )/)

((dump _f _nxt _nxt)/)
((dump _f _fp _nxt) (interm _f _nxt1) (dump _f _nxt1 _nxt) )

((reduceer () () ))
((reduceer (bla|_R) _R1)
     / (reduceer _R _R1))
((reduceer (_x|_R) (_x|_R1))
     (reduceer _R _R1))
P *
```

```
P .
((wrdseq _f "end-of-lex"  "end-of-lex" ))
((wrdseq _f _fp _nxt)
     (wrd _f _fp _nxt1 )/
     (wrdseq _f _nxt1 _nxt))

((wrd _f _fp _nxt)
     (dekken)
     (woord _f _fp _nxt1 _woord)
     (n-of-variants _f _nxt1 _nxt2 _variants)
     (variants _f _nxt2 _nxt _variants _woord))

((woord _f _woord _nxt _woord)
     (interm _f _nxt))

((n-of-variants _f "/" _nxt _x)
     (interm _f _x)
     (LESS 0 _x)
     (interm _f _nxt))
((n-of-variants _f _fp _fp 1))
P .
((variants _f _fp _fp 0 _lex-entry))
((variants _f _fp _nxt _int _lex-entry)
     (syncat _f _fp _nxt1 _CAT)
     (attr-cond-lst _f _nxt1 "*" lex () _attvals)
     (interm _f _nxt2)
     (EL-formule _f _nxt2 ";" _formule)
     (ADDCL
((rule _CAT ((_lex-entry () ())) (() () _attvals ()) _formule))
          )
     (P "^I^B***^D Added rule:" _CAT " --> ^B" (_lex-entry) "^D^M^J")
     (P "^I" _attvals "^M^J^I" _formule "^M^J^M^J")
     (interm _f _nxt3)
     (SUM 1 _XX _int)
     (variants _f _nxt3 _nxt _XX _lex-entry))

((syncat _f _cat _nxt _cat)
     (interm _f _nxt))

P *
P .
((part1 _f _fp _nxt (r|_sels) _opts (_kop|_cats) )
     / (partlabel _f _fp _nxt1)
     (constit-sum _f _nxt1 _nxt2 _sels _cats _opts)
     (resultaat _f _nxt2 _nxt3 _kop)
     (constr-cl _f _nxt3 ";")
     (interm _f _nxt) /)

((constit-sum _f _fp _nxt (_sel|_slst) (_cat|_clst) (_opt|_opts))
     / (constit _f _fp _nxt1 _sel _cat _opt)
```

```
     (rest-van-const-sum _f _nxt1 _nxt _slst _clst _opts) )

((rest-van-const-sum _f "+" _nxt _slst _clst _opts)
     (interm _f _nxt1)
     (constit-sum _f _nxt1 _nxt _slst _clst _opts) /)
((rest-van-const-sum _f "--" _nxt () () ())
     (interm _f ">") (interm _f _nxt) )
P .
((constit _f "(" _nxt _sel (_cat _atts _sem) (_sel _cat))
     (optconst _f "(" _nxt _sel _cat) )
((constit _f _fp _nxt _sel (_cat _atts _sem) bla)
     (const _f _fp _nxt _sel _cat) )

((const _f "[" _nxt _sel _cat)
     (contextconst _f "[" _nxt _sel _cat) )
((const _f _fp _nxt _sel _cat)
     (normalconst _f _fp _nxt _sel _cat) )
P .
((normalconst _f _fp _nxt _sel _CAT)
     (selct _fp _sel) (interm _f :)
     (interm _f _CAT) (interm _f _nxt) )
((optconst _f "(" _nxt _sel _cat)
     (interm _f _nxt1) (const _f  _nxt1 ")" _sel _cat)
     (interm _f _nxt) )
((contextconst _f [ _nxt _sel ([_cat]))
     (interm _f _nxt1) (normalconst _f _nxt1 ] _sel _cat)
     (interm _f _nxt) )
P .
((resultaat _f _fp _nxt _kop)
     (normalconst _f _fp _nxt."r" _kop) )

((constr-cl _f ";" _nxt) /)
((constr-cl _f "[" _nxt)
     (constr-cl1 _f "[" _nxt) )

((constr-cl1 _f "]" _nxt) / (interm _f _nxt) )
((constr-cl1 _f _fp _nxt) (interm _f _nxt1) (constr-cl1 _f _nxt1 _nxt))

P *
P .
((part-II _f _fp _nxt _attr-conds)
     / (part2 _f _fp _nxt1 () _lc)
     (part3 _f _nxt1 _nxt2 _lc _lc-gc)
     (part4 _f _nxt2 _nxt3 (_lc-gc) _lc-gc-gcr)
     (part5 _f _nxt3 _nxt4 _lc-gc-gcr _lc-gc-gcr-icr)
     (part6 _f _nxt4 _nxt _lc-gc-gcr-icr _attr-conds))

((part2 _f _fp _nxt _in _local-conds)
     (partlabel _f _fp _nxt1)
     (attr-cond-lst _f _nxt1 ";" local _in _local-conds)
```

```
     (interm _f _nxt) /)
((part3 _f _fp _nxt _in _conds)
     (partlabel _f _fp _nxt1)
     (attr-cond-lst _f _nxt1 ";" global _in _conds)
     (interm _f _nxt) /)
((part4 _f _fp _nxt _in (_G-carry|_in))
     (partlabel _f _fp _nxt1)
     (carry _f _nxt1 ; _G-carry)
     (interm _f _nxt) /)
((part5 _f _fp _nxt (_G-carry _gc-ic) (_G-carry _I-carry _gc-ic))
     (partlabel _f _fp _nxt1)
     (inci-carry-lst _f _nxt1 ; _I-carry)
     (interm _f _nxt) /)
((part6 _f _fp _nxt (_1 _2 _3) (_1 _2 _newval _3))
     (partlabel _f _fp _nxt1)
     (attr-cond-lst _f _nxt1 ";" new () _newval)
     (interm _f _nxt) / )
p .
((attr-cond-lst _f _nxt _nxt _loc-of-glob-of-new _condities _condities)
     /)
((attr-cond-lst _f _fp _nxt local _in _condities )
     / (attr-cond _f _fp _nxt1 _een-cond)
     (attr-cond-lst _f _nxt1 _nxt local (_een-cond|_in) _condities) )
((attr-cond-lst _f _fp _nxt global _in _condities)
     / (glob-attr-cond _f _fp _nxt1 _een-cond)
     (attr-cond-lst _f _nxt1 _nxt global (_een-cond|_in) _condities) )
((attr-cond-lst _f r _nxt new _in _newval)
     / (new-attr-cond _f r _nxt1 _oneattr)
     (attr-cond-lst _f _nxt1 _nxt new (_oneattr|_in) _newval) )
((attr-cond-lst _f _fp _nxt lex _in _newval)
     / (lex-attr-cond _f _fp _nxt1 _oneatvalpair)
     (attr-cond-lst _f _nxt1 _nxt lex (_oneatvalpair|_in) _newval) )
p .
((attr-cond _f _fp _nxt (_sel (_rel _naam _val)) )
     (const-attr _f _fp _nxt1 _sel _naam)
     (rel _f _nxt1 _nxt2 _rel)
     (value-expr _f _nxt2 _nxt _val) )

((rel _f _fp _nxt on)
    (ON _fp (IN in CONTAINS contains ))
    (interm _f _nxt) )
((rel _f = _nxt eq)
     (interm _f _nxt) )
((rel _f "<" _nxt noteq)
     / (interm _f ">")
     (interm _f _nxt) )
((rel _f _fp _nxt noton)
    (ON _fp (not NOT))
     / (interm _f _nxt1)
    (ON _nxt1 (in IN contains CONTAINS))
```

```
    (interm _f _nxt) )
P .
((glob-attr-cond _f PERM _nxt (_sel _sel1 _naam PERM))
     (interm _f _nxt1)
     (const-attr _f _nxt1 "=" _sel _naam)
     (rel _f "=" _nxt2 _rel)
     (glob-val-expr _f _nxt2 _nxt _sel1 _naam1) )
((glob-attr-cond _f _fp _nxt (_sel _sel1 _naam) )
     (const-attr _f _fp "=" _sel _naam)
     (rel _f "=" _nxt2 _rel)
     (glob-val-expr _f _nxt2 _nxt _sel1 _naam1) )

((new-attr-cond _f _fp _nxt (_naam _val) )
     (const-attr _f _fp = _sel _naam)
     (interm _f _nxt1)
     (value-expr _f _nxt1 _nxt _val) )

((lex-attr-cond _f _naam _nxt (_naam1 _val1) )
     (interm _f _nxt1)
     (value-expr _f _nxt1 _nxt _val)
     /
     (uptolow _naam _naam1)
     (uptolow _val _val1  ))

((uptolow () () ))
((uptolow (_caps|_rest) (_lower|_restlower))
     (uptolow _caps _lower)
     (uptolow _rest _restlower)/)
((uptolow _digit _digit)
     (INT _digit)/)
((uptolow _caps _lower)
     (uptolow1 _caps _lower))

((uptolow1 _CAPS _lower)
     (STRINGOF _C-A-P-S _CAPS)
     (C1 _C-A-P-S _l-o-w-e-r)
     (STRINGOF _l-o-w-e-r _lower))

((C1 () ()))
((C1 (_C|_A-P-S) (_l|_o-w-e-r))
     (Hoofdletter _C)
     (CHAROF _C _aski1)
     (SUM 32 _aski1 _aski)
     (CHAROF _l _aski)
     (C1 _A-P-S _o-w-e-r))
((C1 (_l|_A-P-S) (_l|_o-w-e-r))
     (C1 _A-P-S _o-w-e-r))

((Hoofdletter _char)
     / (NOT LESS _char A)
```

```
     (LESS _char [))
P .
((const-attr _f _fp _nxt _sel _naam)
     (feat-sel _f _fp _nxt1 _sel)
     (attr-naam _f _nxt1 _nxt _naam) )

((feat-sel _f _fp _nxt _sel)
     (selct _fp _sel) (interm _f ".")
     (interm _f _nxt) )

((attr-naam _f _fp _nxt _fp) (interm _f _nxt) )

((glob-val-expr _f _fp _nxt _sel _naam)
     (const-attr _f _fp _nxt _sel _naam) )

((QUOTSTR _fp))

((value-expr _f "[" _nxt _val)
     (interm _f _nxt1)
     (attvallst1 _f _nxt1  "]" () _val)
     (interm _f _nxt) )
((value-expr _f "{" _nxt _vallst)
     (interm _f _nxt1)
     (attval _f _nxt1 _nxt2 _val)
     (attvallst1 _f _nxt2 "}" (_val) _vallst)
     (interm _f _nxt) )
((value-expr _f _fp _nxt _val)
     (attval _f _fp  _nxt _val) )
P .
((attvallst1 _f _fp _fp _vallst _vallst)
     /)
((attvallst1 _f _fp _nxt _tussenlijst _vallst)
     (attval _f _fp _nxt1 _val)
     (attvallst1 _f _nxt1 _nxt (_val|_tussenlijst) _vallst) )

((attval _f , _nxt _val)
     (interm _f _nxt1)
     (attval _f _nxt1 _nxt _val) /)
((attval _f "" _nxt nil)
     (interm _f _nxt) /)
((attval _f _fp _nxt _fp)
     (interm _f _nxt) )
P .
((carry _f _nxt _nxt () ) /)
((carry _f _fp _nxt (_sel))
     (selct _fp _sel) (interm _f "--")
     (interm _f ">" ) (interm _f "r" )
     (interm _f _nxt ) )

((inci-carry-lst _f _nxt _nxt () )/)
```

```
((inci-carry-lst _f _fp _nxt (_lunion| _rest) )
     (inci-carry _f _fp _nxt1 _lunion)
     (inci-carry-lst _f _nxt1 _nxt _rest) )

((inci-carry _f _fp _nxt (UNION _naam _carries) )
     (inci-carry-sum _f _fp r _carries)
     (const-attr _f r _nxt _sel _naam) )

((inci-carry-sum _f _fp _nxt (_sel |_andere-carries) )
     (const-attr _f _fp _nxt1 _sel _naam)
     (rest-van-carry-sum _f _nxt1 _nxt _andere-carries) )

((rest-van-carry-sum _f "+" _nxt _lst)
     (interm _f _nxt1)
     (inci-carry-sum _f _nxt1 _nxt _lst))
((rest-van-carry-sum _f "--" _nxt () )
     (interm _f ">") (interm _f _nxt) )

P *
P .
((part7 _f _fp _nxt _formule) /
     (partlabel _f _fp _nxt1)
     (aandeelhouders _f _nxt1 _nxt2)
     (EL-formule _f _nxt2 ; _formule)
     (interm _f _nxt) )

((aandeelhouders _f ";" _nxt)/)
((aandeelhouders _f _fp _nxt)
      (aandeel _f _fp "'")
      (aandelen _lst )
      (addtolist _lst _fp _res)
      (KILL aandelen)
      (ADDCL ((aandelen _res)) )
      (interm _f _nxt1)
      (rest-van-aandeelhouders _f _nxt1 _nxt) )
P .
((rest-van-aandeelhouders _f + _nxt)
      (interm _f _nxt1)
      (aandeelhouders _f _nxt1 _nxt)/)
((rest-van-aandeelhouders _f "--" _nxt)
      (interm _f ">")
      (interm _f _nxt) )

((aandeel _f _fp _nxt)
      (selct _fp _sel)
      (interm _f _nxt) )

((EL-formule _f ";" ";" ())/)
((EL-formule _f parameter _nxt (parameter _letter))
      (interm _f _letter)
```

```
      (interm _f _nxt) / )
((EL-formule  _f constant _nxt (constant _constant))
     (interm _f _constant)
     (interm _f _nxt) / )
P .
((EL-formule  _f variable _nxt (variable _varnaam))
     (interm _f _varnaam)
     (gebonden-vars _lst) /
     (ON _varnaam _lst) /
     (interm _f _nxt) )
((EL-formule  _f abstraction _nxt
                (abstraction variable _naam _type _descriptor))
     (interm _f variable)
     (interm _f _naam)
     (gebonden-vars _lst)
     (addtolist _lst _naam _res)
     (KILL gebonden-vars)
     (ADDCL ((gebonden-vars _res)) )
     (interm _f _nxt1)
     (semtyp _f _nxt1 _nxt2 _type) /
     (EL-formule _f _nxt2 _nxt _descriptor) /)
((EL-formule _f relation _nxt (relation _rfunxi _arg1 _arg2) )
     (interm _f _rfunxi)
     (interm _f _nxt1) /
     (EL-formule _f _nxt1 _nxt2 _arg1) /
     (EL-formule _f _nxt2 _nxt _arg2) /)
P .
((EL-formule _f element _nxt (element _int _targ) )
     (interm _f _int) /
     (INT _int)
     (interm _f _nxt1) /
     (EL-formule _f _nxt1 _nxt _targ) /)
((EL-formule _f _brancat _nxt (_brancat _1st _2nd _3rd))
     (brct-sort ternary _brancat)
     (interm _f _nxt1) /
     (EL-formule _f _nxt1 _nxt2 _1st) /
     (EL-formule _f _nxt2 _nxt3 _2nd) /
     (EL-formule _f _nxt3 _nxt _3rd) /)
((EL-formule  _f _brancat _nxt (_brancat _arg))
     (brct-sort unary _brancat)
     (interm _f _nxt1) /
     (EL-formule _f _nxt1 _nxt _arg) /)
((EL-formule  _f _brancat _nxt (_brancat _int _formule-lst) )
     (brct-sort lijst _brancat)
     (interm _f _int) /
     (INT _int) /
     (interm _f _fp)
     (EL-formule-lst _f _fp _nxt _int _formule-lst) /)
P .
((EL-formule _f _brancat _nxt (_brancat _agent _object) )
```

```
     (brct-sort ela-ry _brancat)
     (interm _f _agent)
     (interm _f _nxt1) /
     (EL-formule _f _nxt1 _nxt _object) /)
((EL-formule  _f _brancat _nxt (_brancat _1st _2nd) )
     (brct-sort binary _brancat)
     (interm _f _nxt1) /
     (EL-formule _f _nxt1 _nxt2 _1st) /
     (EL-formule _f _nxt2 _nxt _2nd) /)

((brct-sort ela-ry _brct)
     (ON _brct (refgnotion datgnotion refsuspicion datsuspicion
     autogoal allorefgoal allodatgoal) ) /)
((brct-sort ternary _brct)
     (ON _brct (functionvalue conditional) ) /)
((brct-sort lijst _brct)
     (ON _brct (set tuple cartesianpr functionunion) ) /)
((brct-sort binary _brct)
     (ON _brct (application universalqu existentialqu selection
     partselection amount conjunction disjunction unionstar equality
     iteration membership inclusion) ) /)
((brct-sort unary _brct)
     (ON _brct (negation merge power cardinality)) )
P .
((EL-formule-lst _f _fp _fp 0 () ) )
((EL-formule-lst _f _fp _nxt _int (_elt|_rest-elts) )
     (EL-formule _f _fp _nxt1 _elt)
     (SUM _int -1 _int2)
     (EL-formule-lst _f _nxt1 _nxt _int2 _rest-elts) )

((addtolist () _naam (_naam)) /)
((addtolist _lst _naam _lst)
     (ON _naam _lst)/)
((addtolist _lst _naam (_naam |_lst)))

((semtyp _f anytype _nxt anytype)
     (interm _f _nxt))
((semtyp _f atomic _nxt (atomic _string))
     (interm _f _string)
     (interm _f _nxt))
P .
((semtyp _f funtype _nxt (funtype _domain _range))
     (interm _f _nxt1) /
     (semtyp _f _nxt1 _nxt2 _domain) /
     (semtyp _f _nxt2 _nxt _range) /)
((semtyp _f settype _nxt (settype _arg))
     (interm _f _nxt1) /
     (semtyp _f _nxt1 _nxt _arg) /)
((semtyp _f enstype _nxt (enstype _arg))
     (interm _f _nxt1) /
```

```
     (semtyp _f _nxt1 _nxt _arg) /)
((semtyp _f amntype _nxt (amntype _dimension))
     (interm _f _nxt1) /
     (semtyp _f _nxt1 _nxt _dimension) /)
((semtyp _f tuptype _nxt (tuptype _int _ttup) )
     (interm _f _int) /
     (INT _int) /
     (interm _f _fp)
     (ttup _f _fp _nxt _int _ttup) /)
((semtyp _f uniontype _nxt (uniontype _int _ttup) )
     (interm _f _int) /
     (INT _int) /
     (interm _f _fp)
     (ttup _f _fp _nxt _int _ttup) /)
P .
((ttup _f _fp _fp 0 () ) )
((ttup _f _fp _nxt _int (_elt|_rest-elts) )
     (semtyp _f _fp _nxt1 _elt)
     (SUM _int -1 _int2)
     (ttup _f _nxt1 _nxt _int2 _rest-elts) )

P *
P .
((maak-rules () _kop _dochters (_1 _2 _3 _4) _sem)
     (ADDCL ((rule _kop _dochters (_1 _2 _3 _4) _sem)) )
     (P "~M~J~B*** Added PC rule to database:~D~M~J")
     (P "rule          :" _kop " --> " _dochters "~M~J")
     (P "global carry :" _1 "~M~J")
     (P "inci-carry    :" _2 "~M~J")
     (P "new values    :" _3 "~M~J")
     (P "conditions    :" _4 "~M~J")
     (P "semantics     :" _sem "~M~J")
     /)
((maak-rules _opt-paren _kop _dochters _attr-cond-lst _sem)
     (powerset _opt-paren _pow)
     (maak-rules1 _pow _kop _dochters _attr-cond-lst _sem))

((maak-rules1 () _kop _dochters _attr-cond-lst _sem) /)
((maak-rules1 (_deelverz|_r) _kop _dochters (_gc _ic _nv _lc-gc) _sem)
     (pc-variant _deelverz _dochters _dochters1)
     (trim-de-condities 1 _deelverz _gc _gc1)
     (trim-de-condities 1 _deelverz _ic _ic1)
     (trim-de-condities 1 _deelverz _nv _nv1)
     (trim-de-condities 1 _deelverz _lc-gc _lc-gc1)
     (trim-sem 1 _deelverz _sem _sem1)
     / (voeg-rule-toe _kop _dochters1 (_gc1 _ic1 _nv1 _lc-gc1) _sem1)
     (maak-rules1 _r _kop _dochters (_gc _ic _nv _lc-gc) _sem ))
P .
((trim-de-condities _tlr () _acl1 _acl1) /)
((trim-de-condities _tlr ((_sel _cat)) _acl _acl1)
```

```
     / (trim _tlr _sel "}" _acl _acl1))
((trim-de-condities _tlr ((_sel _cat)(_nextsel _cat2)|_r-v-dlverz)  _acl _acl1)
     (trim _tlr _sel _nextsel _acl _acl2)
     (SUM _tlr 1 _tlr1)
     (trim-de-condities _tlr1 ((_nextsel _cat2)|_r-v-dlverz) _acl2 _acl1 ))

((trim-sem _tlr () _formule _formule))
((trim-sem _tlr ((_sel _cat)) _form _form1)
     (trim-sem1 _tlr _sel "}" _form _form1))
((trim-sem _tlr ((_sel _cat)(_nextsel _cat2)|_r-v-deelverz) _f _f1)
     (trim-sem1  _tlr _sel _nextsel _f _f2)
     (SUM _tlr 1 _tlr1)
     (trim-sem _tlr1 ((_nextsel _cat2)|_r-v-deelverz) _f2 _f1))

((trim-sem1 _tlr _sel _nextsel _formule _formule1)
     (trim-sem2 _tlr _sel _nextsel _formule _formule1))
((trim-sem1 _tlr _nextsel _sel _formule () ))

((trim-sem2 _tlr _sel _nextsel ()  () ))
P .
((trim-sem2 _tlr _sel _nextsel ((_x|_r)|_rr) (_lst|_r2) )
     / (trim-sem2 _tlr _sel _nextsel (_x|_r) _lst)
     (trim-sem2 _tlr _sel _nextsel _rr _r2))
((trim-sem2 _tlr _sel _nextsel (_sel|_r) _nouwjazeg )
      / FAIL)
((trim-sem2 _tlr _sel _nextsel (parameter _par) (parameter _sub) )
      (sup _sel _nextsel _tlr _par _sub))
((trim-sem2 _tlr _sel _nextsel (_woord|_r) (_woord|_r1) )
      (trim-sem2 _tlr _sel _nextsel _r _r1))

((trim _tlr _sel _nextsel () () ))
((trim _tlr _sel _nextsel ((UNION _naam (_sel|_sels) )|_R) _result)
     (trim _tlr _sel _nxtsel _R _result))
((trim _tlr _sl _nxtsl ((UNION _naam _r1)|_R) ((UNION _naam _r1res)|_result) )
     / (trim _tlr _sl _nxtsl _r1 _r1res)
     (trim _tlr _sl _nxtsl _R _result))
((trim _tlr _sel _nextsel ((_sel (_relatie _attr _val))|_cnds ) _result)
     (trim _tlr _sel _nextsel _cnds _result))
((trim _tlr _sel _nextsel ((_sel _sel1 _attr)| _cnds )_result)
     (trim _tlr _sel _nextsel _cnds _result))
((trim _tlr _sel _nextsel ((_sel1 _sel _attr)| _cnds )_result)
     (trim _tlr _sel _nextsel _cnds _result))
((trim _tlr _sel _nextsel (_sel|_cnds) _result)
     (trim _tlr _sel _nextsel _cnds _result))
((trim _tlr _sel _nextsel ((_sel _naam)|_cnds)_result)
     (trim _tlr _sel _nextsel _cnds _result))
((trim _tlr _sel _nextsel (_watdanook |_r) (_watdanook1 |_result) )
    (subst _tlr _sel _nextsel _watdanook _watdanook1)
    (trim _tlr _sel _nextsel _r _result))
P .
```

```
((subst _tlr _sel _nextsel  (_sel1 (_relatie _attr _val) )
                            (_sub (_relatie _attr _val) )  )
     (sup _sel _nextsel _tlr _sel1 _sub))
((subst _tlr _sel _nextsel (_sel1 _sel2 _attr)
                           (_sub _sel2 _attr) )
     (sup _sel _nextsel _tlr _sel1 _sub))
((subst _tlr _sel _nextsel (_sel1 _sel2 _attr)
                           (_sel1 _sub _attr) )
     / (sup _sel _nextsel _tlr _sel2 _sub))
((subst _tlr _sel _nextsel (_sel2 _naam) (_sub _naam))
     (sup _sel _nextsel _tlr _sel2 _sub))
((subst _tlr _sel _nextsel (_sel2 _naam) (_sel2 _naam))
     (NOT selct _sel2 _x) /)
((subst _tlr _sel _nextsel _sel2 _sub)
     (sup _sel _nextsel _tlr _sel2 _sub))

((sup _sel _nextsel _tlr _sel1 _sub)
     (LESS _sel _sel1)
     (LESS _sel1 _nextsel)
     (CHAROF _sel1 _aski)(SUM _tlr _aski1 _aski)
     (CHAROF _sub _aski1))
((sup _sel _nextsel _tlr _sel1 _sel1))
P .
((voeg-rule-toe _kop _dochters (_1 _2 _3 _4) _sem)
      (ADDCL ((rule _kop _dochters (_1 _2 _3 _4) _sem)) )
      (P "~M~J~B*** Added PC rule to database:~D~M~J")
      (P "PC-rule       :" _kop " --> " _dochters "~M~J")
      (P "global carry :" _1 "~M~J")
      (P "inci-carry    :" _2 "~M~J")
      (P "new values    :" _3 "~M~J")
      (P "conditions    :" _4 "~M~J")
      (P "semantics     :" _sem "~M~J")
      (P "            ~B***~D~M~J")/)

((pc-variant () _dochters1 _dochters1))
((pc-variant ( (_sel _cat) |_r)  _dochters _dochters1)
     (CHAROF _sel _aski)(SUM 97 _sel-ste _aski)
     (pc-variant _r _dochters _dochters2)
     (verwijder _sel-ste _dochters2 _dochters1))
P .
((verwijder 0 (_cat|_r) _r))
((verwijder _int (_cat|_r) (_cat|_res) )
     (NOT LESS _int 1) (SUM 1 _i _int)
     (verwijder _i _r _res))

((powerset _x _y)
     (lstpower _x ((()) _y) /)

((lstpower () _res _res))
((lstpower (_x|_staart)  _tussen _y)
```

```
   (lstdistribute _x _tussen _res)
   (lstpower _staart _res _y))

((lstdistribute _x (()) ((_x)()) ))
((lstdistribute _x (_y|_R) (_lst _y|_Res) )
     (append _y (_x) _lst)
     (lstdistribute _x _R _Res))
P "*^M^J^M^J"
```

C. De DPSG-parser

```
((parse1 _i () _x _x) / (DUMP _i () (_x) ))

((parse1 _i ((([_cat]) _ats _sem)|_cats) ((_sym _cat)|_rst1)
                                        ((_sym _cat)|_rst ) )
     (strip-tot-rechter-buur _rst1 _rst2)
     (DUMP _i ([_cat]) ((_symb _cat)|_rst1) )
     (parse1 _i _cats _rst2 _rst))
((parse1  _i ((([_cat]) _ats _semuit)|_cats) _words ((? _cat)|_rest))
     (rule (_cat _a _b) _dochters _attsin _semin)
     (DUMP _i (_cat) _words)
     (parse1 ("| "|_i) _dochters _words _words1)
     (eval _dochters _attsin _semin _ats _semuit)
     (strip-tot-rechter-buur _words1 _words2)
     (parse1 _i _cats _words2 _rest))
P .
((parse1  _i ((_cat _ats _sem)|_cats) ((? _cat)|_r1) ((! _cat)|_rest))
     (strip-tot-rechter-buur _r1 _r2)
     (DUMP  _i (_cat) ((? _cat)|_r1) )
     (parse1  _i _cats _r2 _rest))
((parse1  _i ((_term () ())) (_term|_r1) ((! _term)|_rest))
     (strip-tot-rechter-buur _r1 _rest)
     (DUMP  _i (_term) (_term|_r1) )
     (DUMP  _i () _r1 ))
((parse1  _i ((_cat _ats _semuit)|_cats) _words ((! _cat)|_rest))
     (rule _cat _dochters _attsin _semin)
     (DUMP _i (_cat) _words )
     (parse1 ("| "|_i) _dochters _words _words1)
     (eval _dochters _attsin _semin _ats _semuit)
     (strip-tot-rechter-buur _words1 _words2)
     (parse1 _i _cats _words2 _rest))
P .
((strip-tot-rechter-buur () () ))
((strip-tot-rechter-buur ((! _c)|_rest) _rst)
     (strip-tot-rechter-buur _rest _rst)/)
((strip-tot-rechter-buur ((? _c)|_rest) ((? _c)|_rest) )/)
((strip-tot-rechter-buur (_x|_rst) (_x|_rst) ))

((DUMPFILE _i () _invoer ) (dump _dmpfile)
     (Trans _i _ii) (W _dmpfile _ii)
     (DMP3 _dmpfile _invoer) (append _iii ("+--> ") _ii)
     (W _dmpfile _iii) (W _dmpfile ("^M^J") ))
P .
((DUMPFILE _i _lst _invoer ) (dump _dmpfile)
     (W _dmpfile _i) (W _dmpfile _lst)
     (DMP2 _dmpfile _invoer)/)
((DMP1 _dmpfile _invoer ) (W _dmpfile (" ") )
     (flatten _invoer _vlak-inp) (W _dmpfile (_vlak-inp) )
     (W _dmpfile ("^M^J") ))
```

```
((DMP3 _dmpfile _invoer ) (flatten _invoer _vlak-inp)
     (W _dmpfile (_vlak-inp) ) (W _dmpfile ("^M^J") ))
((DMP2 _dmpfile _invoer ) (W _dmpfile (" ") )
     (flatten _invoer _vlak-inp) (W _dmpfile (_vlak-inp) )
     (W _dmpfile ("^M^J") ))
P .
((DUMPSCREEN _i () _invoer ) (Trans _i _ii)
     (px _ii) (DUMP3SCREEN _invoer)
     (append _iii ("+--> ") _ii)
     (px _iii) (P "^M^J" ))
((DUMPSCREEN _i _lst _invoer ) (px _i)
     (px _lst) (DUMP2SCREEN _invoer)/)
((DUMP1SCREEN _invoer ) (P "  " )
     (flatten _invoer _vlak-inp)
     (px (_vlak-inp) ) (P "^M^J" ))
((DUMP3SCREEN  _invoer ) (flatten _invoer _vlak-inp)
     (px (_vlak-inp) ) (P "^M^J" ))
P .
((DUMP2SCREEN _invoer ) (P "  " )
     (flatten _invoer _vlak-inp)
     (px (_vlak-inp) ) (P "^M^J" ))

((flatten _x _res) (flatten1 _x () _res)/)
((flatten1 () _x _x))
((flatten1 (_x|_staart) _result-tot-nu _eindresult)
     (flatten1 _staart _result-tot-nu _result-van-staart)
     (flatten1 _x _result-van-staart _eindresult))
((flatten1 _x _result-tot-nu (_x|_result-tot-nu) ))
P .
((Trans () () ))
((Trans ("+--") ("+--> ") ))
((Trans (_x|_r) (_x|_rr) ) (Trans _r _rr))

((px ())/)
((px (_x|_y))(P _x)(px _y))
((get-ch _a _A _B) (CUWIND _C) (crwind _a 2 55 6 20)
    (px _A) (P "?   ") (FLUSH "TRM:") (GETB "TRM:" _x)
    (CHAROF _B _x) (CUWIND _C) (CLOSE _a))
((get-screen-or-file _x)
     (P "Dump naar het ^Bs^Dcherm of naar ^Bf^Dile?^M^J")
     (get-ch "Uw Keuze" ("^M^J^Bs^Dcherm" "^M^J^Bf^Dile") _x)/)
P .
((pa  _lst) (get-screen-or-file _s) (par S _lst _s))
((par S _lst s) (KILL DUMP)
     (ADDCL ((DUMP |_x)(DUMPSCREEN |_x)) )
     (P "Dump naar het ^Bscherm^D^M^J^M^J")
     (parse S _lst)/)
((par S _lst f) (KILL DUMP)
     (ADDCL ((DUMP |_x)(DUMPFILE |_x)) ) (dump _x)
     (CREATE _x) (OPEN _x)
```

```
     (P "Dump naar file^B" _x "^D^M^J^M^J")
     (parse S _lst)
     (CLOSE _x))
P .
((parse S _y) (parse1 (+--) ((S _a _b)) _y _z) (S-test _z)/)

((S-test ())(P "Zin is ^Bcorrect^D verwerkt^M^J^M^J")/)
((S-test ((? _x)|_rst)) (P "^BMoederloze Constituent!^D ?" _x ) (PP)(PP)/)
((S-test ((! _x)|_rst)) (S-test _rst)/)
((S-test (_x|_r)) (P "Invoer niet geheel verwerkt:" _x "^M^J"))
P *
P "^M^J^BLoading^D EVAL.LOG^M^J."

((eval _dochters ( _gc _ic _nv _lcgc) _semin _atts-uit _sem-uit)
    (eval-lcgc _dochters _lcgc)
    (eval-gc _dochters _gc _gc-uit)
    (eval-ic _dochters _ic _ic-uit )
    (update _gc-uit _ic-uit _nv _atts-uit)
    (eval-sem _dochters _semin _sem-uit))
    ?((/* _nv zijn nieuw en behoeven geen evaluatie))

((eval-sem _dochters () () ))
((eval-sem _dochters ((parameter _sel)|_rest) (_d-sem|_restsem))
     (CHAROF _sel _aski)(SUM 97 _selste _aski)
     (sem-van _selste _dochters _d-sem)
     (eval-sem _dochters _rest _restsem))
((eval-sem _dochters (_x|_sem) (_sem-van-x|_rsem))
     (eval-sem _dochters _x _sem-van-x)
     (eval-sem _dochters _sem _rsem))

((sem-van 0 ((_cat _attrs _sem)|_R) _sem)/)
((sem-van _int ((_cat _a _s)|_r) _sem)
     (NOT LESS _int 1) (SUM 1 _i _int)
     (sem-van _i _r _sem)/)
((sem-van _i () _nouja) FAIL)

((eval-ic _dochters () () )/)
((eval-ic _dochters ((UNION (_sel _naam)) |_rest-unions)
                    ((_naam _value)| _result ) )
     (CHAROF _sel _aski)(SUM 97 _sel-ste _aski)
     (attributen-van _sel-ste _dochters _attrs)
     (get-value _naam _attrs _value)
     (eval-ic _dochters _rest-unions _result ))
((eval-ic _dochters ( (UNION _naam _sels)|_R-unions)
                                    ((_naam _union)| _result ) )
     (verenig-vallijsten _dochters _naam _sels _union)
     (eval-ic _dochters _R-unions _result ))
P .
((eval-lcgc _dochters () )/)
((eval-lcgc _dochters ((_sel1 _sel2 _attr)|_cnds) )
```

```
     (check-gac _dochters _sel1 _sel2 _attr )
     (eval-lcgc _dochters _cnds ))
((eval-lcgc _dochters ((_sel (_rel _naam _val))|_cnds) )
     (check-lac _dochters _sel (_rel _naam _val))
     (eval-lcgc _dochters _cnds ))

((eval-gc _dochters () ())/)
((eval-gc _dochters (_sel) _attrs)
     (CHAROF _sel _aski)(SUM 97 _sel-ste _aski)
     (attributen-van _sel-ste _dochters _attrs))


((update _gc-klaar () () _gc-klaar ))
((update _gc ((_naam NIL)|_rest) _nv _result)
     (update _gc _rest _nv _result))
((update _gc ((_naam _value)|_rest) _nv _result)
     (updeet _gc _naam _value _gc-up)
     (update _gc-up _rest _nv _result))
((update _gc () ((_naam NIL)|_rest) _result)
     (update _gc () _rest _result))
((update _gc () ((_naam _value)|_rest) _result)
     (updeet _gc _naam _value _gc-up)
     (update _gc-up () _rest _result))

((updeet () _naam _value ((_naam _value)) ))
((updeet ((_naam _val1)|_r) _naam _value ((_naam _value)|_r) ))
((updeet (_lst|_r) _naam _value (_lst|_r1) )
     (updeet _r _naam _value _r1))
P .
((check-gac _dochters _sel1 _sel2 _naam )
     (CHAROF _sel1 _aski1)(SUM 97 _sel-ste1 _aski1)
     (attributen-van _sel-ste1 _dochters _attrs1)
     (get-value _naam _attrs1 _val1)
     (CHAROF _sel2 _aski2)(SUM 97 _sel-ste2 _aski2)
     (attributen-van _sel-ste2 _dochters _attrs2)
     (get-value _naam _attrs2 _val2)
     (eq _val1 _val2))

((check-lac _dochters _sel (_rel _naam _val))
     (CHAROF _sel _aski)(SUM 97 _sel-ste _aski)
     (attributen-van _sel-ste _dochters _attrs)
     (get-value _naam _attrs _val1)
     (_rel _val _val1))

((attributen-van 0 ((_cat _attrs _sem)|_R) _attrs)/)
((attributen-van _int ((_cat _a _s)|_r) _attrs)
     (NOT LESS _int 1) (SUM 1 _i _int)
     (attributen-van _i _r _attrs)/)
((attributen-van _i () _nouja) FAIL)
```

```
((get-value _naam () NIL)/ FAIL)
((get-value _naam ((_naam _value)|_rest) _value)/)
((get-value _naam ((_watdanook _eenandere)|_rest) _value)
     (get-value _naam _rest _value)/)
P .
((eq _val _val1)
     (VAR _val)
     / FAIL)
((eq _val1 _val)
     (VAR _val)
     / FAIL)
((eq _val _val1)
     (LST _val)
     (set-eq _val _val1))
((eq _val _val))

((noteq _val _val1)
     (NOT eq _val _val1))

((on _val _lst)
     (ON _val _lst))
((noton _val _lst)
     (NOT ON _val _lst))

((set-eq () () ))
((set-eq () (_x|_R)) /FAIL)
((set-eq (_x|_r) _lst)
     (delete _x _lst _newlst)
     (set-eq _r _newlst))

((delete _x () () )/ FAIL)
((delete _x (_x|_rest) _rest)/)
((delete _x (_y|_rest) (_y|_result))
     (delete _x _rest _result))
P *
P "^L"
P "^M^J^M^J                          ^BDPSG-Parser Loaded^D^M^J^M^J"
P "Usage: pa (string to be parsed)^M^J"
```

D. Voorbeelden van abstracte structuren, gegenereerd door de parser

```
(voorbeeld 1: Gram8.log & Gram8.dmp)
+--S (aa bb cc dd ee ff gg)
|  +--R (aa bb cc dd ee ff gg)
|  |  +--a (aa bb cc dd ee ff gg)
|  |  |  +--[ P ] (bb cc dd ee ff gg)
|  |  |  +--b (bb cc dd ee ff gg)
|  |  |  +--[ c ] (cc dd ee ff gg)
|  |  |  +--d (dd ee ff gg)
|  |  |  +-->(ee ff gg)
|  |  |
|  |  +--c (? c ! d ee ff gg)
|  |  |  +--[ Q ] (ee ff gg)
|  |  |  +--e (ee ff gg)
|  |  |  +--[ f ] (ff gg)
|  |  |  +--g (gg)
|  |  |  +--> ()
|  |  |
|  |  +--f (? f ! g)
|  |  +--> ()
|  |
|  +--P (? P ! c ? Q ! f)
|  +--Q (? Q ! f)
|  +--> ()
|
+---> ()

((rule S (R P Q) ))
((rule R (a ([P]) c ([Q]) f) ))
((rule P (b ([c]) d) ))
((rule Q (e ([f]) g) ))
((rule f ff))
((rule g gg))
((rule a aa))
((rule b bb))
((rule c cc))
((rule d dd))
((rule e ee))

?(((/* voorbeeld: aa bb cc dd ee ff gg))
```

```
(voorbeeld 2: Gram8.log & Gram8.dmp)
+--S (piet eet een appel)
|  +--NPS (piet eet een appel)
|  |  +--NP (piet eet een appel)
|  |  |  +--DET (piet eet een appel)          --> geen DET
|  |  +--NP (piet eet een appel)
|  |  |  +--PROPERNAME (piet eet een appel) --> piet is niet marie
|  |  |  +--PROPERNAME (piet eet een appel) --> piet is niet kees
|  |  |  +--PROPERNAME (piet eet een appel) --> piet is niet paul
|  |  |  +--PROPERNAME (piet eet een appel) --> piet is piet !
|  |  |  |  +--piet (piet eet een appel)
|  |  |  |  +--> (eet een appel)
|  |  |  |
|  |  |  +--> (eet een appel)
|  |  |
|  |  +--[ V ] (eet een appel)
|  |  |  +--eet (eet een appel)
|  |  |  +--> (een appel)
|  |  |
|  |  +--NP (een appel)
|  |  |  +--DET (een appel)
|  |  |  |  +--een (een appel)
|  |  |  |  +--> (appel)
|  |  |  |
|  |  |  +--N (appel)
|  |  |  |  +--appel (appel)
|  |  |  |  +--> ()
|  |  |  |
|  |  |  +--> ()
|  |  |
|  |  +--> ()
|  |
|  +--VP (? V ! NP)
|  |  +--V (? V ! NP)
|  |  +--> ()
|  |
|  +--> ()
|
+--> ()


((rule S (NPS VP)))
((rule NP (DET N)))
((rule NP (PROPERNAME)))
((rule VP (V)))
((rule NPS (NP ([V]) NP)))
((rule PROPERNAME (marie)))
((rule PROPERNAME (kees)))
((rule PROPERNAME (paul)))
((rule PROPERNAME (piet)))
```

```
((rule DET (een)))
((rule N (appel)))
((rule V (eet)))

?(((/* piet eet een appel ))
```

E. De regels en het lexicon uit Aarts e.a. (1988)

E.1. Het Lexicon

```
ZIEN VERB ARGNUM 2 SPEC 1 VFORM INFIN * ZIEN ;
ZAG VERB ARGNUM 2 SPEC 1 VFORM FIN * ZIEN ;
HOREN VERB ARGNUM 2 SPEC 1 VFORM INFIN * HOREN ;
HOORDE VERB ARGNUM 2 SPEC 1 VFORM FIN * HOREN ;
LATEN VERB ARGNUM 2 SPEC 1 VFORM INFIN * LATEN ;
LIET VERB ARGNUM 2 SPEC 1 VFORM FIN * LATEN ;
VOELEN VERB ARGNUM 2 SPEC 1 VFORM INFIN * VOELEN ;
VOELDE VERB ARGNUM 2 SPEC 1 VFORM FIN * VOELEN ;
HELPEN VERB ARGNUM 3 SPEC 2 VFORM INFIN * HELPEN ;
HIELP VERB ARGNUM 3 SPEC 2 VFORM FIN * HELPEN ;
LEREN VERB ARGNUM 3 SPEC 2 VFORM INFIN * LEREN ;
LEERDE VERB ARGNUM 3 SPEC 2 VFORM FIN * LEREN ;
DANSEN VERB ARGNUM 1 VFORM INFIN * DANSEN ;
DANSTE VERB ARGNUM 1 VFORM FIN * DANSEN ;
LEZEN VERB ARGNUM 2 VFORM INFIN * LEZEN ;
LAS VERB ARGNUM 2 VFORM FIN * LEZEN ;
GEVEN VERB ARGNUM 3 VFORM INFIN * GEVEN ;
GAF VERB ARGNUM 3 VFORM FIN * GEVEN ;

AAP NOUN * AAP ;
BOEK NOUN * BOEK ;
NOOT NOUN * NOOT ;

MARTIN PN * MARTIN ;
ASTRID PN * ASTRID ;
ERIK PN * ERIK ;

ZIJ PRON CASUS NOM * ZIJ ;
HIJ PRON CASUS NOM * HIJ ;
HAAR PRON CASUS ACC * ZIJ ;
HEM PRON CASUS ACC * HIJ ;

EEN DET * abstraction
                variable X anytype
                abstraction
                   variable P anytype
                   existentialqu
                      variable X
                      abstraction
                         variable x
                         application
                            variable P
                            variable x ;
DAT TH * abstraction
               variable x anytype
               abstraction
```

```
                variable P anytype
                application
                   variable P
                   variable x ;
```

E.2. De regels

```
C1 { comps }
regel : a:NPS + [b:VERB] + [c:VERB] + d:VERB --> r:COMPS ;
lac : a.ARGNUM = 1
          d.ARGNUM = 2
          d.VFORM = INFIN
          a.CASUS <> NOM ;
gac : ;
gc : ;
ic : ;
nv : r.COMDEPTH = 2 ;
semrul : a' + b' --> application
                        parameter a
                        abstraction
                           variable y anytype
                           abstraction
                              variable x anytype
                              tuple 2
                                 variable x
                                 application
                                    parameter d
                                    tuple 2
                                       variable x
                                       variable y ;
-*-

C2 { comps }
regel : a:NPS + [b:VERB] + [c:VERB] + d:VERB --> r:COMPS ;
lac : a.ARGNUM = 1
          d.ARGNUM = 3
          d.VFORM = INFIN
          a.CASUS <> NOM ;
gac : ;
gc : ;
ic : ;
nv : r.COMDEPTH = 2 ;
semrul : a' + b' --> application
                        parameter a
                        abstraction
                           variable y anytype
```

```
                                abstraction
                                   variable x anytype
                                   tuple 2
                                      variable x
                                      application
                                         parameter d
                                         tuple 3
                                            variable x
                                            element 1 variable y
                                            element 2 variable y ;
-*-

C3 { comps }
regel : a:VERB --> r:COMPS ;
lac : a.ARGNUM = 1
          a.VFORM = INFIN ;
gac : ;
gc : ;
ic : ;
nv : r.COMDEPTH = 1 ;
semrul : a' + b' --> abstraction
                           variable x anytype
                           tuple 2
                              variable x
                              application
                                 parameter a
                                 variable x ;
-*-

C4 { comps }
regel : a:NPS + [b:VERB] + c:VERB --> r:COMPS ;
lac : c.VFORM = INFIN ;
          a.COMDEPTH = NOVAL ;
          a.CASUS <> NOM
          c.SPEC = 2 ;
gac : a.ARGNUM = c.ARGNUM ;
gc : ;
ic : ;
nv : r.COMDEPTH = 1 ;
semrul : a' + b' --> application
                           parameter a
                           parameter c ;
-*-

C5 { comps }
regel : a:NPS + [b:VERB] + c:VERB --> r:COMPS ;
```

```
lac : c.VFORM = INFIN
          a.COMDEPTH = 2
          a.CASUS <> NOM
          c.SPEC = 2 ;
gac : a.ARGNUM = c.ARGNUM ;
gc : ;
ic : ;
nv : r.COMDEPTH = 1 ;
semrul : a' + b' --> application
                          parameter a
                          parameter c ;
-*-

C6 { comps }
regel : a:NPS + b:VERB --> r.COMPS ;
lac : b.VFORM = FIN
          a.COMDEPTH = NOVAL
          a.CASUS = NOM ;
gac : a.ARGNUM = b.ARGNUM ;
gc : ;
ic : ;
nv : r.COMDEPTH = 0 ;
semrul : a' + b' --> application
                          parameter a
                          parameter b ;
-*-

C7 { comps }
regel : a:NPS + b:VERB --> r.COMPS ;
lac : b.VFORM = FIN
          a.COMDEPTH = 1
          a.CASUS = NOM
          b.SPEC = 1 ;
gac : a.ARGNUM = b.ARGNUM ;
gc : ;
ic : ;
nv : r.COMDEPTH = 0 ;
semrul : a' + b' --> application
                          parameter a
                          parameter b ;
semrul : ;
-*-

NPS1 { NPS }
regel : a:NPS + b:NPS --> r.NPS ;
lac : b.COMDEPTH = NOVAL
```

```
          b.ARGNUM = 2 ;
gac : ;
gc : ;
ic : r.CASUS = a.CASUS ;
nv : r.COMDEPTH = NOVAL
          r.ARGNUM = 3 ;
semrul : a' + b' --> abstraction
                          variable R anytype
                          application
                             parameter a
                             abstraction
                                variable x anytype
                                application
                                   parameter b
                                   abstraction
                                      variable y anytype
                                      application
                                         variable R
                                         tuple 3
                                            variable x
                                            element 1 variable y
                                            element 2 variable y;
-*-

NPS2 { NPS }
regel : a:NPS + b:NPS --> r.NPS ;
lac : b.COMDEPTH <> NOVAL
          b.ARGNUM = 2 ;
gac : ;
gc : ;
ic : r.COMDEPTH = b.COMDEPTH
          r.CASUS = a.CASUS ;
nv : r.ARGNUM = 3 ;
semrul : a' + b' --> abstraction
                          variable R anytype
                          application
                             parameter a
                             abstraction
                                variable x anytype
                                application
                                   parameter b
                                   abstraction
                                      variable y anytype
                                      application
                                         variable R
                                         tuple 3
```

```
                                                variable x
                                                element 1 variable y
                                                element 2 variable y ;
-*-

NPS3 { NPS }
regel : a:NP + b:NP --> r.NPS ;
lac : ;
gac : ;
gc : ;
ic : r.CASUS = a.CASUS ;
nv : r.COMDEPTH = NOVAL
          r.ARGNUM = 2 ;
semrul : a' + b' --> abstraction
                        variable R anytype
                        application
                           parameter a
                           abstraction
                              variable x anytype
                              application
                                 parameter b
                                 abstraction
                                    variable y anytype
                                    application
                                       variable R
                                       tuple 2
                                          variable x
                                          variable y ;
-*-

NPS4 { NPS }
regel : a:NP --> r.NPS ;
lac : ;
gac : ;
gc : ;
ic : r.CASUS = a.CASUS ;
nv : r.COMDEPTH = NOVAL
          r.ARGNUM = 1 ;
semrul : a' + b' --> parameter a ;
-*-

NPS5 { NPS }
regel : a:NP + b:COMPS--> r.NPS ;
lac : b.COMDEPTH <> 2 ;
gac : ;
gc : ;
```

```
ic : r.CASUS = a.CASUS ;
nv : r.ARGNUM = 2
          r.COMDEPTH = 2 ;
semrul : a' + b' --> abstraction
                          variable R anytype
                          application
                             parameter a
                             tuple 2
                                abstraction
                                   variable x anytype
                                   abstraction
                                      variable P anytype
                                      application
                                         variable P
                                         parameter b
                                abstraction
                                   variable y anytype
                                   application
                                      variable R
                                      tuple 2
                                         variable x
                                         variable y;
-*-

NPS6 { NPS }
regel : a:NP + b:COMPS--> r.NPS ;
lac : b.COMDEPTH = 2 ;
gac : ;
gc : ;
ic : r.CASUS = a.CASUS ;
nv : r.ARGNUM = 2
          r.COMDEPTH = 2 ;
semrul : a' + b' --> abstraction
                          variable R anytype
                          application
                             variable R
                             application
                                parameter a
                                parameter b ;
-*-

NP1 { NP }
regel : a:PN --> r.NP ;
lac : ;
gac : ;
gc : ;
```

```
ic : ;
nv : ;
semrul :  a' + b' --> abstraction
                          variable P anytype
                          application
                             variable P
                             parameter a ;
-*-

NP2 { NP }
regel : a:DET + b:NOUN --> r.NP ;
lac : ;
gac : ;
gc : ;
ic : ;
nv : ;
semrul :  a' + b' --> application
                          parameter a
                          parameter b ;
-*-

NP3 { NP }
regel : a:PRON --> r.NP ;
lac : ;
gac : ;
gc : ;
ic : r.CASUS = a.CASUS ;
nv : ;
semrul : a' + b' --> abstraction
                         variable P anytype
                         application
                            variable P
                            parameter a ;
-*-

NP4 { NP }
regel : a:TH + b:COMPS --> r.NP ;
lac : b.COMDEPTH = 0 ;
gac : ;
gc : ;
ic : ;
nv : ;
semrul : a' + b' --> application
                         parameter a
                         parameter b ;
-*-
```

```
end-of-rules
```

Referenties

[1] Aarts, E., M. Kammler en A. Tops (1987) *De behandeling in DPSG van gekruiste afhankelijkheden in Nederlandse bijzinnen.* Werkstuk, KUB (FLW)

[2] Bunt, H.C., J. Thesing en K. van der Sloot (1987) *Discontinuous constituents in trees, rules and parsing.* Tendum Technical Report No. 2. Computational Linguistics Unit, Tilburg University and Institute for Perception Research/IPO, Eindhoven.

[3] Bunt, H.C., R.J. Beun, F.J.H. Dols, J.A. van der Linden en G.O Schwartzenberg. (1985) *The* TENDUM *system and its theoretical basis.* IPO annual progress report 19; Institute for Perception Research/IPO, Eindhoven.

[4] Bunt, H.C. (1985) *Model-theoretic semantics and Mass terms.* Cambridge University Press, Cambridge.

[5] Bunt, H.C. (1988) *Discontinuous Phrase Structure Grammar and its use in sentence generation.* In: Advances in Natural Language generation, Vol. 2; M. Zock and G. Sabah eds. Printer Publishers, London.

[6] Eijck, J. van. (1986) *Logica en formele taalkunde.* Syllabus FLW, kenmerk 822.86.D25.

[7] Jensen, K. en N. Wirth (1974 en 1976) *Pascal User Manual and Report.* Revised for the ISO pascal Standard 2nd ed. Springer Verlag, Berlin.

[8] McGabe, F.G., K.L. Clark en B.D. Steel. (1984) *Micro Prolog 3.1. Programmers Reference Manual.*

[9] Togt, H. van der. (1988) *Dynamische interpretatie van anaforen in TENDUM* Doctoraalscriptie. KUB (FLW)

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS